HIGH-LEVEL LANGUAGE COMPILING FOR USER-DEFINABLE ARCHITECTURES.

Donald Carlton Simoneaux

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

High-Level Language Compiling
For User-Definable Architectures

by

Donald Carlton Simoneaux

Thesis Advisor: V. M. Powers

Approved for public release; distribution unlimited.

June 1975

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) High-Level Language Compiling For User-Definable Architectures		5. TYPE OF REPORT & PERIOD COVERED Electrical Engineer June 1975
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) LT Donald Carlton Simoneaux		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1975
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

High-level language, compiler, hardware design language, Microcomputer programming, PL/M, configuration-independent compiler, software engineering, code optimization, modularity

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The development of large-scale integrated circuits in the last few years has resulted in a rapid increase in the number of digital devices available for electronic system design. Skilled system designers often do not have an abundance of software experience and require better tools than are presently available in order to take maximum advantage of microprocessors and other functional building blocks. A case is made for the development of a high-level language compiler which will allow the designer to specify not only his algorithm but also his hardware configuration and his optimization

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73
(Page 1)   S/N 0102-014-6601 |

Block #20.

constraints. The PL/M compiler developed by Intel Corporation is used as
a model for examining some of the requirements of this "machine-
independent" compiler. A summary of work which was done to implement the
first stages of such a compiler is presented, and factors which must be
considered in order to further this work are discussed.

High-Level Language Compiling
For User-Definable Architectures

by

Donald Carlton Simoneaux
Lieutenant, United States Navy
B.S.E.E., Tulane University, 1967
M.S.E.E., Naval Postgraduate School, 1974

Submitted in partial fulfillment of the
requirements for the degree of

ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL
June 1975

ABSTRACT

The development of large-scale integrated circuits in
the last few years has resulted in a rapid increase in the
number of digital devices available for electronic system
design. Skilled system designers often do not have an abun-
dance of software experience and require better tools than
are presently available in order to take maximum advantage
of microprocessors and other functional building blocks. A
case is made for the development of a high-level language
compiler which will allow the designer to specify not only
his algorithm but also his hardware configuration and his
optimization constraints. The PL/M compiler developed by
Intel Corporation is used as a model for examining some of
the requirements of this "machine-independent" compiler. A
summary of work which was done to implement the first stages
of such a compiler is presented, and factors which must be
considered in order to further this work are discussed.

TABLE OF CONTENTS

LIST OF FIGURES

7

I. INTRODUCTION

A. PROBLEM DEFINITION

The most promising and widely discussed device in the electronics industry during the last few years has been the microprocessor. This device packages the central processing unit and associated elements of a digital computer into a handful of integrated circuit chips; in many cases only one chip is used. Since the advent of the microprocessor in 1971 it has become much easier to incorporate the power of a digital computer into the design of an electronic system. Compared with custom Large Scale Integration (LSI) circuits microprocessors are convenient, flexible, low-cost devices which have allowed sophisticated features to be made available in relatively simple systems. As a result their use has expanded rapidly, and many people who have had limited programming experience are now being forced to write programs as part of their design efforts.

The term "firmware" has come to be used for systems which utilize programmable digital components, since the development of such systems requires both hardware and software design. The design of a firmware system, whether it uses a microprocessor or some other means of providing a programming capability, is a complex task requiring the best skills of both the electronics engineer and the computer programmer.

Another development which, although conceived in the early 1950's, has become significant only in the last decade is the use of microprogramming in digital systems. Microprogramming differs from "normal" programming primarily in the level of detail considered. Each instruction in the instruction set of a typical digital computer requires several hardware operations to be performed, but these operations are transparent to the programmer. In microprogrammable systems each of these primitive operations may be invoked by a microinstruction. Initially, as in the IBM System/360, microprogramming was done only by the manufacturer, but today there are general purpose computers (e.g., the Hewlett-Packard HP-2100 and Burroughs "D" machine) which allow user microprogramming. In addition there are proposals for using standard functional modules in the implementation of special purpose digital systems [56]. These modular systems will be controlled by what amount to microprograms.

As in the case of the microprocessor, microprogrammed systems will in most instances be programmed by engineers who have a firm background in hardware design but who may have minimal software experience. Thus it is becoming increasingly necessary that programming languages be developed which are easy to use and which can produce good control code for a variety of architectures. The compiler for such a language could be considered a software computer aided design (CAD) tool for the engineer. Ideally, it would

10

accept a description of the algorithm to be performed (the program) and descriptions of the hardware and the format of the control code; the output would then be a control program to perform the algorithm.

Succeeding chapters of this thesis examine some of the considerations necessary in the development of a programming language for firmware system design. Chapter II contains a discussion of programming languages and the advantages and disadvantages of high-level languages. The language PL/M is presented in Chapter III and is used as a basis for examining the necessary features of a high-level language. The implementation of pass 1 of a PL/M compiler is described in Chapter IV. This chapter also describes some of the theoretical aspects of programming language design and implementation. The output of pass 1 is an intermediate language representation of a source language program, an important concept which is discussed in Chapter V.

A major factor in the implementation of any digital system is the system architecture. Chapter VI contains descriptions of various types of architectures and a discussion of the influence of architecture on language design. Optimization of the output code is another important consideration in the design of a compiler. Many firmware systems will be produced in large numbers, and the amount of hardware used will have a significant impact on the cost. Because of the fierce competition among manufacturers, good optimization techniques will be critical in the

11

implementation of these systems. Chapter VII is devoted to the topic of compiler optimization.

The topics discussed in Chapters IV-VII are tied together in Chapter VIII, which shows how they all influence the design of a compiler for user-definable architectures. Chapter IX summarizes the conclusions reached during the study of the problem and presents a list of recommendations for future work.

B. SOFTWARE ENGINEERING

With the rapidly growing use of digital techniques in electronic system design has come the emergence of a new discipline, that of software engineering, to address problems at the hardware-software interface. Although digital computers have been in existence for more than 30 years, it is only today becoming widely recognized that the software design considerations are at least as important as the hardware design considerations in digital system development [11]. The acceptance of the fact that software problems are of more than academic interest is highlighted by the recently inaugurated publication of a new technical journal--the IEEE Transactions on Software Engineering. Because software engineering addresses many issues which are very closely related to the firmware design problem, its goals and principles--as defined by Ross, Goodenough, and Irvine [51]--are outlined below. These ideas have a strong influence on much of the remainder of this thesis.

The goals of software engineering are:

1) Modifiability--This refers to the ability to make con-
   trolled changes in a program. In a large system
   software modifications have to be made during develop-
   ment as well as after production has begun. Modifica-
   tions may be made either to correct errors or to change
   or add features and provide varying levels of perfor-
   mance (i.e., a "family" of systems).

2) Efficiency--This goal is concerned with the best utili-
   zation of the resources available. Typically this
   means using the least memory and time in performing the
   task. Efficiency is usually "... prematurely permitted
   a high priority in engineering tradeoffs ... [but] does
   not dominate the practice of software engineering."
   [51, p.20-21]

3) Reliability--This is a critical goal, especially for
   software systems used in real-time control applica-
   tions. Unfortunately reliability has too often in the
   past been considered as secondary to efficiency in
   software development.

4) Understandability--This goal supports the goals of
   modifiability and reliability. If a piece of software
   is easily understandable, it is easy to modify and easy
   to check for reliability. It is unfortunate that un-
   derstandability is usually considered to reduce effi-
   ciency, but this relationship does not necessarily
   hold. Increased understandability can lead to the
   detection of inefficiencies in a large system.

13

There are seven principles of software engineering which
may be applied in order to achieve the goals. These princi-
ples are:

1) Modularity--This refers to the purposeful structuring
   of a system. Modularity is an important principle in
   both hardware and software design.

2) Abstraction--The unessential details are omitted at any
   given level in the design, leaving only abstract con-
   cepts for consideration.

3) Localization--Limiting the scope of a structure or a
   concept is closely related to modularity. Localization
   enhances confirmability and understandability.

4) Hiding--"... [T]he purpose of hiding is to make inac-
   cesible certain details that should not affect other
   parts of a system." [51, p.22]

5) Uniformity--It is important that definitions and con-
   cepts be applied uniformly across a system.

6) Completeness--Specifying all details and leaving noth-
   ing to chance greatly increases reliability.

7) Confirmability--This refers to the ability to determine
   whether all the design goals have been met.

Software engineering is concerned with the question of
whether it is more important to have very efficient code, in
the sense that it uses the minimum amount of memory and exe-
cutes at the maximum rate (two goals which, by the way, are
usually not compatible), or whether it is more important to
have code which is reliable, easy to modify, and takes a

minimum amount of time to develop.   As   in   all   engineering

disciplines,   software   engineering   is involved with making

tradeoffs among   the   various   alternatives,   since   no   one

answer will be correct in all situations.

II. <u>PROGRAMMING</u> <u>LANGUAGES</u>

In the early days of digital computer use it became evident that an alternative to machine language programming was needed. A computer program is really nothing more than a series of binary digits contained in some storage medium, but human engineering dictated that early machine language programs be represented as groups of octal digits. With the introduction of mnemonics and assemblers to translate them, programs became almost readable. Assemblers became more and more sophisticated with the addition of macros, comment fields, and conditional assembly features, but programs still were tedious to write and difficult to read. The drawback of assembly language programs is that they contain too much information about the operation of the hardware (contrary to the principle of abstraction), and this tends to obscure information related to the algorithm being implemented. Since there is essentially a one-to-one correspondence between assembly language instructions and machine instructions, assembly language programs still tend to be very cumbersome and error-prone except when used for very simple problems. Thus, as programs became increasingly complex, high-level languages were introduced.

A. THE CASE FOR HIGH-LEVEL LANGUAGES

The development of high-level languages was spurred by the desire to be able to write programs which are more

16

descriptive of the problems being solved and which depend
less on the actual hardware on which the programs are to
execute. "High-order languages represent a concept for
improving the understandability of programs by abstracting
from the details of computer instruction sets." [51, p.19]
These languages are designed to facilitate description of
the procedural steps involved in problem solution, and thus
they are often referred to as procedure-oriented languages
(as opposed to machine-oriented assembly languages).

The main advantages of programming in a high-level
language are:

1) The programmer is freed from the consideration of many
   minor details. These details are mainly in the nature
   of bookkeeping--memory allocation, register allocation,
   assignment of temporary variables to hold the results
   of partial computations, remembering branch locations,
   type checking of variables, and many others. This fac-
   tor is becoming even more important with the increased
   use of microprogrammable systems. "The inability of a
   user to cope with a highly intricate, time-and-machine
   dependent environment often results in inefficient, if
   not error prone, microprograms." [47, p.791]

2) Efficient control structures greatly reduce the burden
   of programming, resulting in increased reliability.

3) Symbolic user variables increase the readability of the
   program. This is also one of the advantages assembly
   languages have over machine languages.

17

4) The ability to write arbitrary arithmetic expressions also increases readability and tends to reduce computational errors.

5) Programmer productivity is increased because of the expansion factor involved in the translation from high-level language to machine language. It is generally recognized that programmers produce, on average in a large project, only a few lines of code per day, whether it be machine code, assembly code, or high-level language code.

6) Documentation is improved, because the program is more understandable. A good high-level language encourages the writing of programs which are essentially self-documenting.

7) Maintenance, modification, and debugging are facilitated because of the improved readability and documentation.

8) Transportability is improved, because a high-level language has little dependence on a particular machine architecture. In fact one goal of language design is complete machine independence. This topic is covered more fully in Chapter VIII.

By far the most popular criticism of high-level languages is based upon the concern for efficiency. There are basically two sources of inefficiency. The first has to do with the fact that in certain instances some languages are too machine independent in that they do not recognize

18

features which are basic to computer hardware. For example, FORTRAN does not contain primitive operations for bit manipulation (shifting, rotating, masking, etc.). A good high-level language should not restrict the programmer from doing anything that he could do at the assembly or machine language levels.

The second source of inefficiency lies within the code generation process and is really a characteristic of the compiler rather than the language. The complaints most often voiced by those opposed to the use of a high-level language are that the compiler generates too much code and that the code generated is wasteful of time. However, the point is usually demonstrated with only a small program [19,43].

These inefficiencies are really local in nature, since each instance can usually be isolated to a few lines of code. A good assembly language programmer can write locally "optimal" code, but in a large program his code will suffer from global inefficiencies (see advantage (1) above). Thus "... data based upon comparisons between small programs will tend to underestimate the advantage of the higher level language for large programs." [23, p.214] Many large programs written in high-level languages would have been very difficult and costly to write in assembly language [34] and probably would have been less efficient.

It is doubtful whether any compiler will ever be able to generate completely locally optimal code (as compared with

19

assembly language versions), but there are many promising techniques emerging (see Chapter VII). Experience has shown that global inefficiency is a nonlinear function of program length, and a good compiler can usually produce more efficient code than an assembly language programmer for programs longer than about 50 to 100 high-level language statements [23]. For shorter programs an engineering decision must be made as to whether the advantages of programming in a high-level language outweigh the loss in efficiency. A more complete discussion of this topic is presented in Section VII.A. As memory costs continue to fall, the extra code generated by the local inefficiencies in a compiler will take on lesser significance even in small system design projects.

B. SYSTEM PROGRAMMING LANGUAGES

An area very closely related to firmware design is that of system programming. For many years system programmers have avoided the use of high-level languages, and for the same reason that the designers of programmable hardware (firmware) are now avoiding them--ineffiency. In addition to the fact that software engineering considerations are causing this position to be reevaluated, many advances have been made in the past few years in the area of programming language design. The development of good, machine-independent, high-level languages for system programming has been studied for several years [21,40], and a few languages have been implemented.

The UNIX operating system [50], designed for the popular PDP-11 series of minicomputers, is currently in use at more than 50 installations around the country (including the Computer Science Laboratory at the Naval Postgraduate School) and was written almost entirely in the C language [49], an Algol-like high-level language. In fact, the C compiler itself was written in C. The fact that an interactive, multi-user operating system as sophisticated as UNIX can be implemented satisfactorily on a minicomputer confirms the viability of high-level language programming in a situation requiring efficient machine code.

C. COMPOSITE LANGUAGES

One alternative solution to the problem of choosing between a high-level language and an assembly language is the composite language--a language which has (hopefully) the best features of both. The simplest implementation is a high-level language which allows assembly code to be inserted into a program. PL/360 is an example of this type of language. The advantage of using such a language is claimed to lie in in the ability to make use of the efficiency of the assembly language while retaining the benefits of high-level language programming.

Aside from the loss of understandability there are two major disadvantages in using this approach. First is the loss of machine independence, which reduces transportability. Each time the architecture of the hardware is changed (e.g., by using a different processor or by rearranging a

21

modular system), the program must be carefully examined for instructions which need to be changed. The second disadvantage is the reduction in reliability brought about by the fact that the programmer is allowed access to facilities which normally are completely controlled by the compiler. This can lead to conflicts (e.g., in resource allocation) and may cause unexpected results and subtle side effects which are difficult to trace.

These disadvantages were partially overcome in the implementation of the Language for Systems Development (LSD) [40]. In LSD the use of assembly language is restricted to macros whose definitions are separate from the program itself. Except for the fact that the notation involved seems somewhat clumsy, this approach probably comes very close to the ideal notion of a machine-independent compiler.

A slightly different approach was taken by Popper [43] in the implementation of SMAL, which is in essence an assembly language with some of the structure of a high-level language. A SMAL program equivalent to the example presented in Section VII.A was written by Popper, and it required only four more bytes of memory than the assembly language version. Although the SMAL version is easier to read than the assembly language version, it is more difficult to read than the PL/M version.

Composite approaches such as Popper's probably will be very beneficial for programmers who are designing small microprocessor systems but they do not appear to provide the

22

best long-range solution to the firmware design problem. Succeeding sections will make it evident that compiler theory is advancing to the point of favoring the development of high-level languages which do not allow such highly machine-dependent features as are found in composite languages.

III. THE PL/M LANGUAGE

In the effort to provide comprehensive software support for its eight-bit microprocessors, Intel Corporation was led naturally in 1973 to the development of the high-level language PL/M [29,53]. Since then several other microprocessor manufacturers have announced either the availability or the anticipated availability of PL/M compilers (with possibly some slight modifications) for their microprocessors. The first large scale application of the language by Intel, ironically, was in the development of a sophisticated macro-assembler to run on its Intellec 8 microcomputer developmental system.

PL/M is derived from the XPL compiler-writing language [42], which in turn is a derivative of PL/I. Thus PL/M is very closely related to both of these languages in its syntax and semantics. A complete list of the syntactic productions is given in the Intel reference manual [29], and the syntax and semantics of the C language version used for this investigation is given in Appendix B (see file "m.gram"). It should be noted that the syntax for the C version is not written in the standard BNF notation but rather in the notation required by YACC (see Section IV.B.1).

There have been many proposals over the years for the development of machine-independent programming languages (e.g., MPL [18], which is also similar to XPL). PL/M,

although not currently machine-independent, has the advantage of having been implemented and used for practical system development. Thus PL/M was chosen as the vehicle for examining some of the considerations in the development of a machine-independent high-level language for firmware system design. The remainder of this section is devoted to a brief description of the language and a discussion of its advantages and possible shortcomings.

A. LANGUAGE FEATURES

Lloyd and Van Dam have defined a high-level language to be one which has the following features:

(1) Symbolic user variables (allocated by the compiler),
(2) Ability to evaluate arbitrary arithmetic or logical expressions,
(3) Flow of control statements beyond simple (conditional and unconditional) GOTO, SKIP, Branch and Link. [38, p.537]

In his search for a high-level programming language, Eckhouse found the need for one that was "... procedural, descriptive, flexible, and possibly machine-independent." [17, p.169]. PL/M has all of these features, including a limited kind of machine-independence. The latter feature is exhibited in the ability of PL/M programs to be compiled for either the 8008 or the 8080 microprocessor. Although these two devices are both manufactured by Intel and have somewhat similar instruction sets, they have different architectures and a significant difference in the flexibility and speed of execution of their instructions. These points will be explored further in Chapters VII and VIII.

25

As is its predecessors, PL/M is a block-structured language with a comprehensive set of control structures. "... [T]he control structures of sequential flow, conditional selection, and iteration are sufficient to implement any algorithm." [60, p.35] Sequential flow is provided by the simple statement and the DO-END group, while conditional selection is accomplished by three constructs: IF-THEN and IF-THEN-ELSE statements and DO CASE groups. The DO FOR group is used for a fixed number of iterations, and the DO WHILE group is used for iterating until some condition is satisfied. The GOTO statement is also provided in PL/M for use in those rare circumstances where the use of the other control structures may be somewhat awkward. In recent years considerations of software engineering have discouraged indiscriminate use of the GOTO since "... goto-free programming forces programmers to make explicit the conditions under which a given statement is executed, and this can help ensure understandability and prevent errors." [51, p.21]

PL/M is relatively easy to learn and read and has a simple character set. This latter factor may be important, since a language intended for use in a wide variety of design environments should not require special character sets such as those of APL or some of the proposed microprogramming languages (e.g., see [47]). Ease of learning and readability are important in increasing programmer productivity and program modifiability and reliability.

In order to give a more complete picture of the features of PL/M, a sample program [29] is presented in Figure 1.

26 .

```
1.     2048:   /* is the origin of this program */
2.declare tto literally '2', cr literally '15q',
3.         lf literally '0ah',
4.         true literally '1', false literally '0';
5.
6.squareroot: procedure(x) byte;
7.     declare (x,y,z) address;
8.     y =x; z = shr(x+1,1);
9.         do while y <> z;
10.         y = z; z = shr(x/y + y + 1, 1);
11.         end;
12.     return y;
13.     end squareroot;
14.
15.print$char: procedure(char);
16.     declare bit$cell literally '91',
17.         (char,i) byte;
18.     output (tto) = 0;
19.     call time (bit$cell);
20.         do i = 0 to 7;
21.         output(tto) = char; /* data pulses */
22.         char = ror(char,1);
23.         call time(bit$cell);
24.         end;
25.     output(tto) = 1;
26.     call time (bit$cell + bit$cell);
27.     /* automatic return is generated */
28.     end print$char;
29.
30.print$string: procedure(name,length);
31.     declare name address,
32.         (length,i,char based name) byte;
33.         do i = 0 to length - 1;
34.         call print$char(char(i));
35.         end;
36.     end print$string;
37.
```

Figure 1. Sample PL/M program
for computing square roots
(continued on next page)

```
38.print$number: procedure(number,base,chars,zero$suppress);
39.    declare number address,
40.           (base,chars,zero$suppress,i,j) byte;
41.    declare temp (16) byte;
42.    if chars > last(temp) then chars = last(temp);
43.        do i = 1 to chars;
44.        j = number mod base + '0';
45.        if j > '9' then j = j + 7;
46.        if zero$suppress and 1 <> 1 and number = 0 then
47.            j = ' ';
48.        temp(length(temp)-i) = j;
49.        number = number / base;
50.        end;
51.    call print$string(.temp+length(temp)-chars, chars);
52.    end print$number;
53.
54.declare i address,
55.    crlf literally 'cr,lf',
56.    heading data (crlf,lf,lf,
57.    '                        table of square roots',
58.    crlf,lf,
59.    ' value  root value  root value  root value  root',
60.    ' value  root',
61.    crlf,lf);
62.
63.    /* silence tty and print computed values */
64.    output(tto) = 1;
65.    do i = 1 to 1000;
66.    if i mod 5 = 1 then
67.        do; if i mod 250 = 1 then
68.            call print$string(.heading,length(heading));
69.        else
70.            call print$string(.(cr,lf),2);
71.        end;
72.    call print$number(i,10,6,true /* true suppresses
73.                                leading zeroes */);
74.    call print$number(square$root(i),10,6, true);
75.    end;
76.
77.declare monitor$uses (10) byte;
78.eof
```

Figure 1 (continued). Sample PL/M program
for computing square roots (after [29])

This program (as well as most other PL/M and C programs reproduced in this thesis) is written in lower case characters, since this is the normal input mode for the UNIX operating system, which was used for all of the work described. In addition to the features previously mentioned, notice should be taken of the comment convention of the language. Since comments can be placed anywhere within a program (rather than on separate lines as in FORTRAN), self-documentation is encouraged. Although the "/* */" convention is a little awkward, it has the advantage of setting off comments and not discouraging short comments (as does the "COMMENT" convention in ALGOL).

Figure 2 presents a second sample PL/M program which demonstrates another significant feature of the language-- the nested macro-definition capability. While the macro-definition concept is certainly not new, there are many languages which do not allow macros (most notably FORTRAN and ALGOL). Many languages which do have a macro capability do not allow nesting. As can be seen, the macros increase the readability of the program, but there is another, perhaps greater, advantage in using them. By using macros the programmer can specify certain items only once in a program (e.g., vector sizes and input/output ports) and then use the macro names elsewhere in the program to refer to those items. Later he can modify his program by merely changing the appropriate macro definitions. While the advantages of being able to do this are not as evident in the

```
1./* paper tape reader controller program */
2.
3.declare forever literally 'while 1',
4.          cdata    literally 'output(1)',
5.          cstat    literally 'output(2)',
6.          ccom     literally 'input(2)',
7.          ccps     literally 'input(3)',
8.          rdata    literally 'input(1)',
9.          rstat    literally 'input(0)',
10.         rcom     literally 'output(0)',
11.         noreq    literally 'not ccom',
12.         acon     literally 'ror(rstat,1)',
13.         perr     literally '10b',
14.         badcps   literally '100b',
15.         ok       literally '> 3 and cps < 26',
16.         rrdy     literally 'rstat',
17.         clk1     literally '1b',
18.         clk0     literally '0b',
19.         drdy     literally '1b';
20.
21.declare cps byte,
22.         wait(22) byte initial
23.              (250,200,167,143,125,111,100,91,83,77,71,
24.               67,63,59,56,53,50,48,45,43,42,40);
25.
26.do forever;
27.    cstat = 0;
28.
29./* wait until read request */
30.    do while noreq;
31.    end;
32.
33./* determine the characters per second rate */
34.    cps = ccps;
35.
36.    if acon and rrdy then
37.        do;
38.        if cps ok then   /* we are ready */
39.            do;                /* to read characters */
40.            cdata = rdata;
41.            rcom = clk1;
42.            rcom = clk0;
43.            cstat = drdy;
44.            cstat = 0;
45.            /* wait for tape to move */
46.            call time(wait(cps - 4));
47.            end; else cstat = badcps;
48.        end; else cstat = perr;
49.end;
50.eof
```

Figure 2. Another sample PL/M program

short program of Figure 2 as they would be in a large pro-
gram, it should be apparent that this will increase the
modifiability, and consequently the reliability, of programs
written in the language.

Another, less significant, factor which increases reada-
bility is the inclusion of the separator "$" in some of the
long identifiers in the program of Figure 1. This character
is ignored by the scanner in the PL/M compiler when included
within identifiers and numbers.

Examination of the PL/M manual [29] and the programs in
Figures 1 and 2 will reveal that PL/M contains functions
which relate directly to the Intel 8008 and 8080 instruction
sets. Thus PL/M fits the definition given by Lloyd and Van
Dam for a "tailored" language: "A language whose features
are explicitly designed to coincide (to a large extent) with
the hardware capabilities of its object machine ...."
[38, p.540] Fortunately this is not as serious a drawback as
it might seem, as evidenced by the fact that other microproc-
essor manufacturers are now developing or have developed
PL/M compilers for their machines. All of the functions in
PL/M which relate specifically to the 8008 and the 8080 are
implemented as built-in functions; i.e., they are equivalent
to procedures (and variables, in some cases) which are de-
clared in an encompassing block level hidden from the pro-
grammer. Lloyd and Van Dam [38] recognized that this is an
important concept, and the method by which it is implemented
is explained further in Chapter IV.

31

The built-in function approach is probably preferable in firmware applications to the extensible language approach, although this will probably be a topic of considerable debate for many years. An extensible language is essentially one which has a more sophisticated macro capability than PL/M. It allows the programmer to define new instructions and redefine the base instructions of the language. This may seem to be a desirable feature, but unfortunately it violates the principle of uniformity. Halstead observed that "... the extensible-language approach ... seemed to open the door to a dangerous, undisciplined proliferation of overlapping and even incompatible dialects within a single installation ...." [23, p.214]

B. POTENTIAL MODIFICATIONS

In order for PL/M to serve as a useful general-purpose, machine-independent programming language for firmware design, it will probably be necessary to make some slight modifications. The changes described below were suggested by study of other programming languages which are similar in structure to PL/M, with particular attention being paid to the C language [49]. This language has relative merits and shortcomings when compared with PL/M, but it is a good system programming language which generates efficient machine code for the PDP-11 series of minicomputers. Most of the items listed below are convenience features rather than necessities. (Of course, a major advantage of high-level languages is their convenience when compared with assembly

32

languages.) Many of them were not implemented in the original versions of PL/M, probably because they tend to lead to less efficient machine code, but most of them would not be difficult to implement and some might even allow more efficient code to be generated. The optimization techniques discussed in Section VII.B would be of benefit for those features with apparent inefficiencies.

One major weakness of PL/M is its paucity of data types. If the language were to be used as the basis of a firmware design system, it would need at least a concept of floating point variables in order to be widely accepted. Other desirable data types include string and substring, bit, double precision, and complex. It would also be convenient to have the capability to define data structures. Implementation of some of these various data types would probably suggest the need for a few new instructions for manipulating them efficiently. For example, double precision arithmetic instructions and string concatenation instructions would be useful.

For algorithms involving array arithmetic it would be desirable to have the capability to declare arrays of dimension greater than one. A related feature is the ability to declare arrays with variable lower bounds, as in ALGOL W.

Recursion is another feature which PL/M lacks; however, this may not be significant for firmware design applications. Recursion allows compact expression of an algorithm but is not a necessary feature in a language, since a recursive procedure may be rewritten as an iterative procedure.

Besides, recursion usually sacrifices execution efficiency for programming efficiency, and great care must be taken in writing recursive procedures in order to ensure that they do not "blow up."

A feature which would prove very useful, especially in large system development, is the ability to link independently compiled and tested segments of a program. The current Intel versions of the PL/M language do not allow this, since the second pass of each compiler produces absolute machine code. The implementation of this feature would require the declaration of "global" or "external" variables, the redesign of pass 2 to produce relocatable object code, and the design of a linking loader.

As will be discussed in Chapter VI, the trends in digital architecture development have encouraged, among other features, inclusion of multiple high-speed registers and fast increment/decrement instructions. One way in which to allow the high-level language programmer to take advantage of such features is to provide special constructs within the language. For example, he could be allowed to declare frequently referenced variables to be "register" variables in order to increase execution speed (and also produce a slight saving in the amount of main memory required). The programmer could also be allowed to write statements such as

$$i = ++j - k;$$

or

$$i = j-- - k;$$

in order to take advantage of the increment and decrement

34

instructions. The first statement above would generate code to increment "j," subtract the value of "k" from the new value of "j," and store the result in "i"; while the second statement would generate code to subtract the value of "k" from the value of "j," store the result in "i," and then decrement "j."

Both the register declaration and increment/decrement features are available in the C language. The increment/decrement feature should be really just a convenience for the programmer, since the same statements could be written in C as

$$i = (j = j + 1) - k;$$

and

$$i = j - k; \quad j = j - 1;$$

and the compiler should generate the same code as for the previous two statements (unfortunately it doesn't--see Section VII.B.1). The register declaration in C does result in more efficient code being generated; however, there may be other ways to solve the register allocation problem. This point is discussed further in Section VII.B.2.

Another potential change in PL/M would involve the addition of the conditional expression. This would enable the statement

$$if \ a < b \ then \ c = a; \ else \ c = b;$$

to be rewritten more concisely as

$$c = if \ a < b \ then \ a \ else \ b;$$

and could be done by merely adding a few more productions to

the grammar (see Section IV.B.1). This change would not increase the efficiency of the generated code.

One final area for potential modification involves the CASE statement and will probably require a great deal more study than some of the changes suggested above. The CASE construct in PL/M can be awkward and error-prone in some situations, as illustrated in Figure 3(a). It should be noted that it would have been very easy for the programmer to incorrectly count the number of semicolons in this section of code. Also he has had to resort to the much-maligned GOTO in order to share code between two of the cases, and his only control over an out-of-range value of "c" is to make a test before entering the CASE group.

Figure 3(b) shows how the same routine would be implemented if the C language SWITCH statement were available in PL/M. In this "case" there is no need to count semicolons. The use of the GOTO is avoided, since the cases may be listed in any order, and the BREAK is used to exit from the group. Also there is a specific default case to ensure that appropriate action is taken for all values of "c."

Both the CASE and the SWITCH constructs have advantages and disadvantages in comparison with one another. The CASE statement, despite the drawbacks noted above, will produce more efficient code in many situations and should not be discarded in favor of the SWITCH. Ross, et al, highlighted one of the tradeoffs involved when they noted that

   ... to ensure completeness of case statement control a
   programmer should be permitted by the syntax to specify
   what should happen when a case statement variable is out

36

```
-----------------------------------------------------------------


if c < 'a' or c > 'u' then call err;
    do case c - 'a';
    ;;;
    togd = true;      /* case 'd' */
    ;;;;;;;
        do;           /* case 'l' */
        l = true;
        go to lab1
        end;
    ;;;
    togp = true;      /* case 'p' */
    ;;;
    togt = true;      /* case 't' */
    lab1: do;         /* case 'u' */
        lim = 0;
            do while (c := getc) >= '0' and c <= '9';
            lim = lim * 10 + c - '0';
            end;
        if l then liml = lim;
        else limu = lim;
        end;
    end /* case group */;


                          (a)


    do switch c;
    case 'd': togd = true; break;
    case 'l': l = true;
    case 'u': lim = 0;
            do while (c := getc) >= '0' and c <= '9';
            lim = lim * 10 + c - '0';
            end;
        if l then liml = lim;
        else limu = lim; break;
    case 'p': togp = true; break;
    case 't': togt = true; break;
    default: call err;
    end;


                          (b)


    Figure 3.  (a) PL/M code using the CASE construct,
        (b) PL/M code using the SWITCH construct


-----------------------------------------------------------------
```

of range. Confirmability applied to the same issue would
imply a programmer should be <u>required</u> to state what should
happen. Of course, if he knows that out of range values
are not possible, this too should be expressible, to per-
mit implementation efficiency. [51, p.23]

Vaughn [58] has suggested that both facilities could be pro-

vided in the same language in the form of a generalized IF

statement. A simpler alternative would seem to be to incor-

porate the structure of Figure 3(b) into the present PL/M

language along with the CASE statement.

IV. PASS 1 IMPLEMENTATION

Aho and Ullman [3] have suggested that the process of compilation is composed of seven subprocesses: lexical analysis, error analysis, bookkeeping, parsing, translation (to an intermediate form), code optimization, and object code generation. While it may be difficult to identify all seven of these subprocesses in any given compiler and their order may not be the same as that given, this is a good conceptual model. Figure 4 shows how each of the parts of this model is related to the others [3, p.74].

This chapter documents the initial stages of the design of a compiler for user-definable architectures. All but the code optimization and code generation phases of this model were implemented. The latter two phases are discussed in Chapters VII and VIII, and suggestions are given there for their implementation. Recommendations for continuation of the design are given in Chapter IX.

A. THE FORTRAN VERSION

In order to gain insight into the analysis of some of the problems presented in other sections of this thesis, it was felt that some practical experience in compiler implementation was desirable. For reasons given in Chapter III the PL/M microprocessor language was chosen for this purpose, and an attempt was made to implement the commercial
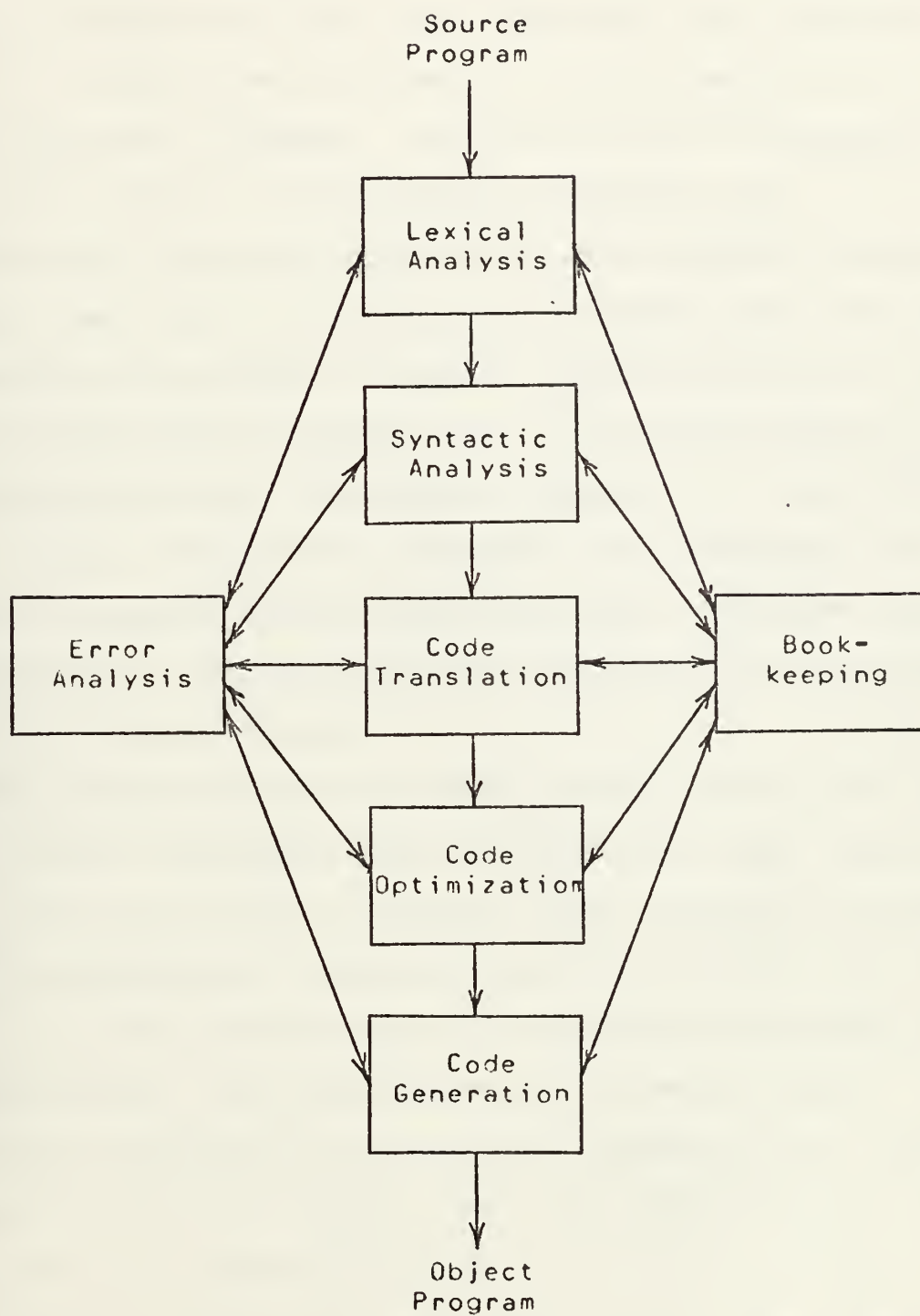
Figure 4. Model of a compiler (after [3])

FORTRAN version on a Digital Equipment Corporation PDP-11/50 computer with an interactive operating system. Unfortunately this proved unfeasible, and attention was shifted to writing another version of pass 1 of the compiler in a system programming language. The latter effort was successful, and a full account is given below in Section IV.B.

The main reason for the failure of the FORTRAN implementation was that it required more primary memory than was available on the PDP-11. When run on the IBM S/360 at the Naval Postgraduate School, pass 1 of the PL/M compiler requires approximately 120K bytes of memory. On the PDP-11 only about 56K bytes of user memory were available, and if all of the object modules could have been linked and loaded together it is estimated that they would have occupied about 100-110K bytes of memory.

An attempt was made to divide the routines in such a way that several sub-passes could be generated, each requiring less than 56K bytes; however, there turned out to be too much interdependence among the routines, and there was always at least one partition which required more memory than was available. This was because the synthesis routine (the one which generates the intermediate language code) required about 50K bytes by itself, and it required many other routines to be loaded with it.

Another problem which developed involved the discovery that the data initialization statements in the Intel PL/M compiler do not conform to ANSI standard FORTRAN

41

specifications (although it is claimed that the compiler is written in standard FORTRAN to enhance transportability). The FORTRAN compiler used for this project accepts only programs written in standard FORTRAN. It requires each variable initialized in a DATA statement to be named individually, and this caused problems with the vast number of vectors which are initialized in the BLOCK DATA routine. This by itself was not a critical problem and could have been overcome without too much difficulty if there had been justification to continue working with the FORTRAN version.

After the first attempts to partition pass 1 of the compiler failed, there were two alternatives available. Either a more concerted effort could have been made to subdivide the FORTRAN version, or a completely new version could have been attempted in a more efficient language. After considering the amount of effort which would be involved in working with the FORTRAN version and the inherent inefficiencies entailed in running it on a 16-bit machine (e.g., it assumes 32-bit integers) it was decided that it would be simpler and more beneficial in the long run to write another compiler.

B. THE C VERSION

After the FORTRAN version was abandoned, pass 1 of the PL/M compiler was successfully implemented using the compiler writing facilities supported by the UNIX operating system [50]. Since a secondary objective of the project was to develop a system for experimenting with compiler design, it proved worthwhile to utilize these more efficient

facilities. Because of the time constraints placed upon this project only pass 1 of the compiler was implemented; however, much valuable experience was gained in the process, and a great deal of this thesis has been influenced by the results obtained.

1.   YACC

For many years compiler writing was more of an art than a science, but many important developments have taken place over the last decade to reverse this situation.   Some of the most impressive of these developments have been in the area of formal language theory and automatic parser generation.   "The ability to generate parsers from a syntactic description of a language is an important consideration in reducing the cost of developing reliable translators." [60, p.34]

The parser generator in the UNIX system is known as YACC (Yet Another Compiler-Compiler) [30]. It has been in use for about two years at Bell Laboratories where, among other things, it has been utilized in the development of an easy-to-use language for a sophisticated mathematics typesetting system [32]. Input for YACC consists of a syntactic and semantic description of the grammar of the language for which a parser is desired.   Appropriate languages belong to the class known as LALR [2,3,4], or look-ahead LR, since they read text from the left, perform a right-parse, and resolve conflicts by looking ahead in the text stream.   This is a very broad and useful subset of the

43

context-free languages, and one which includes PL/M. YACC checks the grammar for conflicts and, if none exist, produces a set of parse tables for the language.

The semantics associated with each production in the grammar are transformed by YACC into a C program which contains the parse tables as data. When this program has been compiled it is linked with a parse table interpreter, provided by the YACC library, and any other programs which have been written by the compiler designer. Actually YACC provides only the core of the compiler--the parser (which performs the syntactic analysis function of Figure 4) and a means of communication between the parse stacks and the programs provided to perform the other functions (lexical analysis, error analysis, bookkeeping, and code translation) of the code generation process.

File "m.gram" in Appendix B contains the YACC input for PL/M. The syntactic notation is somewhat different from the normally encountered BNF, in which a production might be written as

        <NONTERM1> ::= <NONTERM2> TERMINAL <NONTERM3>

rather than the YACC version

        nonterm1:   nonterm2 'terminal' nonterm3

in which all terminal symbols are quoted unless they have been declared to be terminals (as have "identifier", "number", and "string" on the first line of "m.gram"). The convention of using a vertical bar ("|") to indicate the beginning of a production with the same left side as the

44

immediately preceeding production has been retained from BNF. The semicolon (";") is used to indicate the end of a set of productions with the same left side. It should be noted that a quoted semicolon ("';'") may occur within a production as a terminal symbol.

Semantics are provided by appending an equal sign ("=") followed by a C language statement (compound statements are enclosed in braces, "{" and "}") to a production before either the vertical bar or the semicolon. The procedures used for implementing the semantics of PL/M are discussed in Section IV.B.5.

The extreme flexibility afforded by the use of an automatic parser generator such as YACC is demonstrated in Figure 5, which shows the changes required in the PL/M grammar in order to implement the conditional expression construct (see Section III.B). Productions 86 and 87 are currently included in the compiler implemented for this project, and productions 87a-87c are the new productions which would have to be added.

```
    expression: logicalexpression      /* 86 */
    !      variable ':' '=' logicalexpression /* 87 */
    !      ifexpression          /* 87a */
    ;
    ifexpression: trueobject expression      /* 87b */
    ;
    trueobject: ifclause expression 'else'      /* 87c */
    ;
```

Figure 5. Potential syntax changes for adding
the conditional expression to PL/M (see Chapter III)

45

2. Data Structures

One of the first and most important steps in design-
ing a complex software system is the definition of an ap-
propriate set of data structures. The principal structures
used in the implementation of the PL/M compiler are
described below in order to give a fuller understanding of
the nature of the problem and insight into the changes which
would be necessary in order to expand the use of the com-
piler to a more general environment. The declarations of
all of the stacks and tables used can be found in the file
"m.decl" in Appendix B. The macros used in the declarations
are defined in file "m.def."

The two most important data structures used in a
modern compiler are the parse stack and the symbol table.
The parse stack in an LALR parser is used to store input
tokens for the "shift" and "reduce" operations. In general,
there are at least two parallel stacks which contain various
pieces of information about the tokens. YACC provides parse
stacks in its parse table interpreter routine, with one
stack being reserved for values provided by the scanner.
The operation of these stacks is rather complex and will not
be considered here, since Aho and Johnson [2] have provided
an excellent survey of the techniques involved. Since there
was a need to retain more than a single piece of information
about each token, and there was no way to communicate with
the parse stacks in the parse table interpreter other than
to provide a single value, it was necessary to implement

46

four other stacks for this purpose. The operation of these stacks is discussed in Section IV.B.3.

The symbol table is important for a number of reasons, not the least of which is the fact that it is used in conjunction with the intermediate language output to transmit information to the later passes of the compiler. It usually accounts for the bulk of the main memory data storage requirements of the compiler and must therefore be implemented in as efficient a manner as possible.

The symbol table is a vector of eight-bit bytes which, during the course of a compilation, consists of a series of entries of varying types. The format of a general symbol table entry for the PL/M compiler is shown in Figure 6. This is the type of entry which is generated for all variables and procedures declared by the programmer. Reserved words and macro definitions also are represented by symbol table entries, but the formats of these entries are slightly different from that shown in Figure 6. The differences are described below, following the description of the general type.

The first three bytes of the format are common to all three types of entries and are referred to as fixed information ("finfo" in the programs). The first byte contains the "last" field, which specifies the number of bytes to the beginning of the preceding entry and is used for chaining downward through the table (as, e.g., when printing or dumping the symbol table). It should be noted that since

47

```
+-----------------------------------+
|                                   |
|              length               |
|                                   |
|   -  -  -  -  -  -+-----------+    |
|                  |           |     |
|              length          |     |
|                  |           |     |
|   +--------------------+     |     |
|                                   |
|              syno                 |
|                                   |
+-----------------------------------+
|                                   |
|                                   |
|              hcoll                |
|                                   |
|                                   |
+-----------------------------------+
|                                   |
|                                   |
|                                   |
|                                   |
|              name                 |
|                                   |
|                                   |
|                                   |
|                                   |
+-----------------------------------+
|                                   |
|              size                 |
|                                   |
+-----------------------------------+
|     |          |                  |
|  b  |   prec   |      type        |
|     |          |                  |
+-----------------------------------+
|                                   |
|              last                 |
|                                   |
+-----------------------------------+
```

Figure 6.
Format of a general symbol table entry
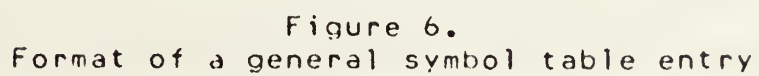
the "last" field contains only eight bits each symbol table
entry is limited to 256 bytes, although it will generally be
much shorter. This in turn ultimately limits the lengths of
variable and procedure names and macro definitions, since
the characters for describing these attributes must fit into
the remainder of the entry after the fixed information and
other fields have utilized some of the 256 bytes.

The second byte of the entry contains three fields:
"type," "precision (prec)," and "based (b)." The "type"
field consists of four bits and is used to distinguish among
the various types of entries (variable, reserved word, mac-
ro, vector, etc.). The alternatives can be found in file
"m.def." The precision field contains three bits and is most
commonly used to represent the precision (i.e., the number
of bytes required) of variables, vectors, or the result of a
function procedure call (zero indicating no value returned).
The "based" field, if set to "1," indicates a based, or
indirect, variable.

Next is the "size" field, in the third byte. This
field is used to indicate the length of the following two
fields, "name" and "hcoll." The "name" field contains the
printname of the symbol and has a length equal to the number
of characters in the name. The "hcoll" field is always two
bytes long and contains the absolute address of the previous
symbol table entry whose printname has the same hash code as
this symbol (see Section IV.B.3). Thus the value of "size"
is equal to the length of the printname plus two.

49

During the course of this project it was found that compiler-generated labels were the only entries which had no printname, and the entries for these symbols conveyed no information other than the symbol number (see below). Since they only took up precious symbol table space (especially for long programs, which already require a great deal of space), these entries were eliminated from the symbol table. Examination of the symbol table in Figure 8 makes it evident which symbol numbers are used for compiler-generated labels, since these are the only symbols which do not have entries (e.g., S25, S28, S29).

Following the "hcoll" field in the general symbol table entry is the "syno" field. This field contains the symbol number of the entry. Each time a new symbol is declared by the programmer an entry of this type is made, and the next sequential symbol number is assigned. The "syno" field is ten bits long, and thus there can be as many as 1024 different symbols in any program (including compiler-generated labels).

The final field in the general entry is the "length" field, which indicates the number of elements in a vector or the number of arguments required by a procedure. In the latter case "length" may be zero, for a procedure with no arguments, or as large as 63, since a procedure definition uses only the first six bits of this field. (This restriction could easily be changed; however, it is doubtful whether any procedure in a well-written program would have more than 63 arguments.)

A saving of table space is accomplished by classify-
ing vectors into two categories, short and long, depending
upon whether or not they contain fewer than 64 elements. In
the case of short vectors (distinguished from long vectors
by the "type" field) and variables, the byte containing the
last eight bits of the "length" field is deleted, as dis-
cussed in Section IV.B.3.

Figure 7 shows the changes required in the general
format of Figure 6 for reserved words, macro definitions,
and based variables. As indicated, all fields from "last"
through "hcoll" remain as in the general format. Figure
7(a) indicates that the entry for a reserved word (e.g.,
"do," "for," "while") has one additional byte, the "resno"
field, containing the reserved word number, which is impor-
tant in the parsing process. Since this field contains only
eight bits, there can be no more than 256 reserved words in
the language. Following the "hcoll" field in the entry for
a macro definition (Figure 7(b)) are the "msize" and "mdef"
fields, the former giving the number of characters in the
definition (restricted to a maximum of 255) and the latter
containing the definition. For a based variable the "based"
field contains a "1," and there is a "bsyno" field inserted
between the "syno" field and the "length" field, as shown in
Figure 7(c). The "bsyno" field contains the symbol number
of the variable which serves as the base. The six unused
bits in this type of entry are wasteful, but most programs
do not contain many based variables.
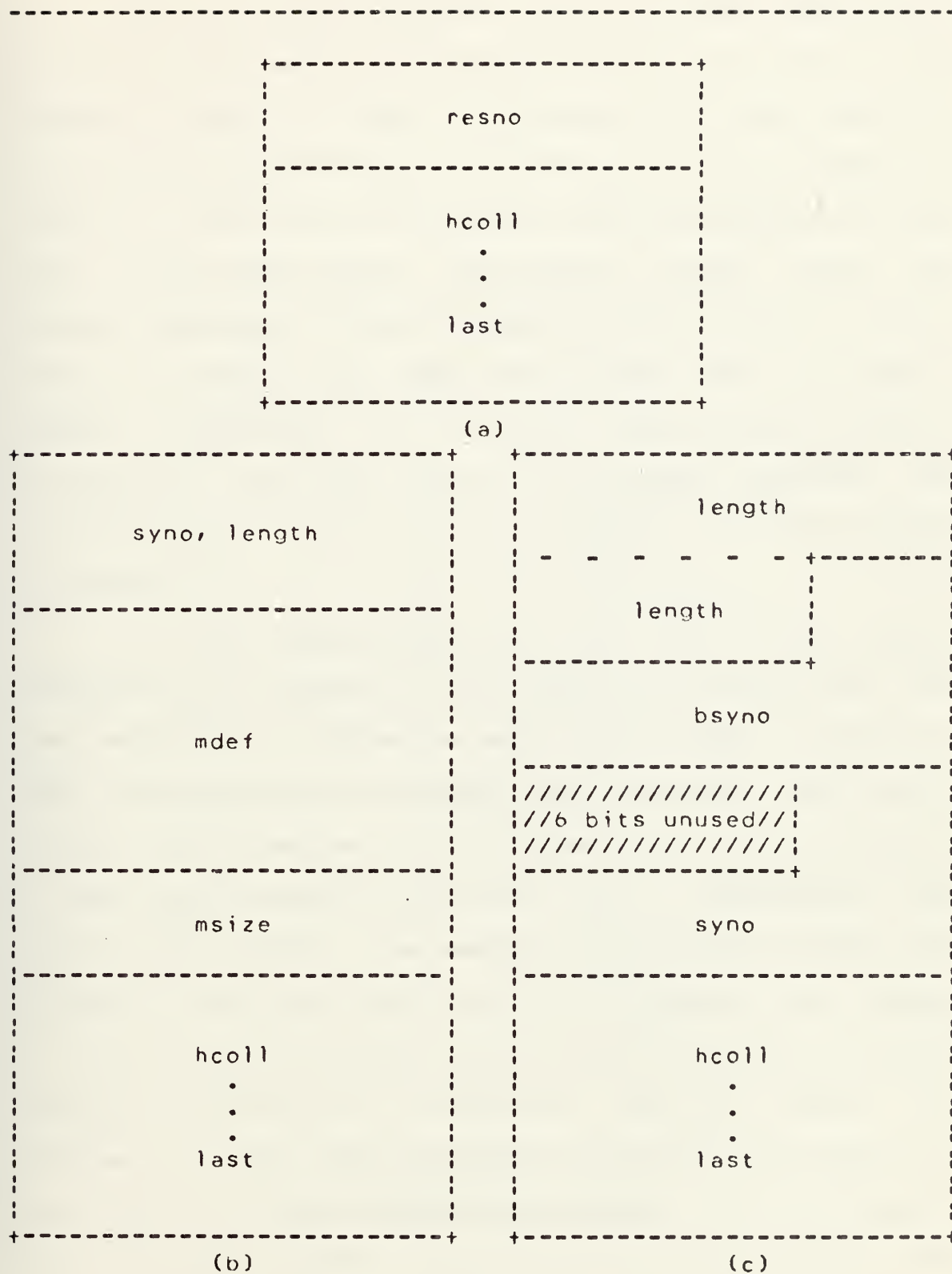
```
        +-------------------------------+
        !            resno              !
        !- - - - - - - - - - - - - - - -!
        !            hcoll              !
        !              .                !
        !              .                !
        !              .                !
        !            last               !
        +-------------------------------+
                       (a)
+-----------------------+  +-----------------------+
!                       !  !          length       !
!      syno, length     !  !- - - - - - +----------!
!                       !  !   length   !          !
!- - - - - - - - - - - -!  !- - - - - - +          !
!                       !  !          bsyno         !
!                       !  !-----------------------!
!         mdef          !  !///////////////////!   !
!                       !  !//6 bits unused//!     !
!                       !  !///////////////////+    !
!- - - - - - - - - - - -!  !- - - - - - - - - - - -!
!         msize         !  !          syno         !
!- - - - - - - - - - - -!  !- - - - - - - - - - - -!
!         hcoll         !  !          hcoll        !
!           .           !  !            .          !
!           .           !  !            .          !
!           .           !  !            .          !
!         last          !  !          last         !
+-----------------------+  +-----------------------+
          (b)                        (c)

           Figure 7.  Format modifications for
       (a) reserved words, (b) macro definitions,
                 (c) based variables
```

52

Now that the various fields in a symbol table entry have been explained, it should prove useful to look at an example. Figure 8 shows the symbol table which was constructed by the PL/M compiler for the square root program of Figure 1. Each line of the printed table corresponds to one entry in the symbol table. The reserved words, which are stored immediately below symbol S0, are not shown in this table. It should be noted that there are no "syno," "based," "precision," or "length" field entries for macro definitions. The "name" column of the table contains the printnames of all entries and the "msize" and "mdef" fields for macros.

A very important point to note here is that symbols S0-S22 were not declared in the square root program but are the variables and procedures which relate PL/M to the Intel 8080. These symbols were placed into the symbol table during the initialization of the compiler and can be considered to have been declared in an outer block encompassing the square root program. The manner in which this was done can be seen by examining file "m.main.c" in Appendix B. Since it is very easy to change the names and attributes of these symbols in "m.main.c" it is also very easy to tailor the language to the architecture of the machine for which the object code is to be generated (see Chapter III). The meanings of these symbols need not be of concern during the first pass of the compilation but will be important during later passes.

Syno	B	Pr	Len	Type	Size	Name
S67		1	10	12	13	monitoruses
S60		1	98	11	9	heading
				1	6	crlf 5 cr,lf
S58		2	1	2	3	i
S52		1	16	12	6	temp
S51		1	1	2	3	j
S50		1	1	2	3	i
S48		1	1	2	14	zerosuppress
S47		1	1	2	7	chars
S46		1	1	2	6	base
S45		2	1	2	8	number
S44		0	4	6	13	printnumber
S41	★	1	1	2	6	char Based S37
S40		1	1	2	3	i
S38		1	1	2	8	length
S37		2	1	2	6	name
S36		0	2	6	13	printstring
S33		1	1	2	3	i
				1	9	bitcell 2 91
S31		1	1	2	6	char
S30		0	1	6	11	printchar
S27		2	1	2	3	z
S26		2	1	2	3	y
S24		2	1	2	3	x
S23		1	1	6	12	squareroot
				1	7	false 1 0
				1	6	true 1 1
				1	4	lf 3 0ah
				1	4	cr 3 15q
				1	5	tto 1 2

Figure 8. PL/M symbol table
for the program of Figure 1

Syno	B Pr	Len	Type	Size	Name
S22	2	1	7	0	
S21	2	2	8	5	dec
S20	2	2	8	8	double
S19	1	1	8	6	move
S18	1	1	8	6	last
S17	1	1	8	8	length
S16	1	1	8	8	output
S15	1	1	8	7	input
S14	1	1	8	5	low
S13	1	1	8	6	high
S12	0	1	8	6	time
S11	1	2	8	5	scr
S10	1	2	8	5	scl
S9	1	2	8	5	shr
S8	1	2	8	5	shl
S7	1	2	8	5	ror
S6	1	2	8	5	rol
S5	2	1	7	10	stackptr
S4	1	1	7	8	memory
S3	1	1	7	8	parity
S2	1	1	7	6	sign
S1	1	1	7	6	zero
S0	1	1	7	7	carry

Figure 8.   PL/M symbol table
for the program of Figure 1 (continued)

3. The Parser

The main function of pass 1 of the PL/M compiler is to convert the source language program into a form which can be used by remaining stages of the compiler to generate machine code. The source language program is represented in the computer as a linear string of ASCII characters, organized as a series of "identifiers," "numbers," and "strings" (all called "tokens"). This series of tokens is the "text stream" for the compiler. In order to perform the translation, pass 1 must parse the program; i.e., it must examine the text stream and determine which of the rules of the PL/M grammar can be applied in order to reduce the tokens to a "statement list" and finally to a "program" (see file "m.gram" in Appendix B). This section contains an overview of the parsing and symbol table functions of pass 1.

When the parser provided by YACC requires a token from the text stream, it calls the user-provided routine "yylex." (In this section "user" refers to the compiler designer rather than the firmware designer.) This routine and the routines which it calls are listed in file "m.scan.c" (Appendix B). "yylex" calls "gettoken," which constructs tokens from the input characters, determines which of the three types of tokens (or a special character--e.g., comma, semicolon) it has found, and computes a hash code for each identifier. The latter function is accomplished by forming the sum, modulo 128, of the ASCII values of the characters in the printname of the identifier.

This hash code is used by "yylex" later for looking up the identifier in the symbol table.

The vector "varc" (Figure 9) is used by "gettoken" to accumulate characters from the input string. Several tokens may be accumulated in "varc" before being used by the parser, and the variable "tokindex" is used to indicate the element of "varc" which is the beginning of the current "accumulator." The first byte of each accumulator contains the length of the token, thus limiting the length of each token to no more than 254 characters. Since the length of "varc" is normally less than 255, and it may contain more than one token, the upper bound on the length of a token is usually much less than 254.

Once "gettoken" has completed its functions, control returns to "yylex," which may take one of several sets of actions, depending on the type of token scanned. If an end of file character or other special character was scanned, "yylex" returns the character to the parser. If a number was scanned, "yylex" reports this to the parser and returns the value of the number. If either a string or an identifier was scanned, "yylex" "pushes" information onto the user-controlled parsing stacks (Figure 9). (The stack manipulation routines are listed in file "m.aux.c.") In the case of a string, the stack pointer ("sp") is incremented, "var[sp]" is assigned the current value of "tokindex," and "tokindex" is advanced to the value of the next free location in "varc." The fact that a string was scanned is

57
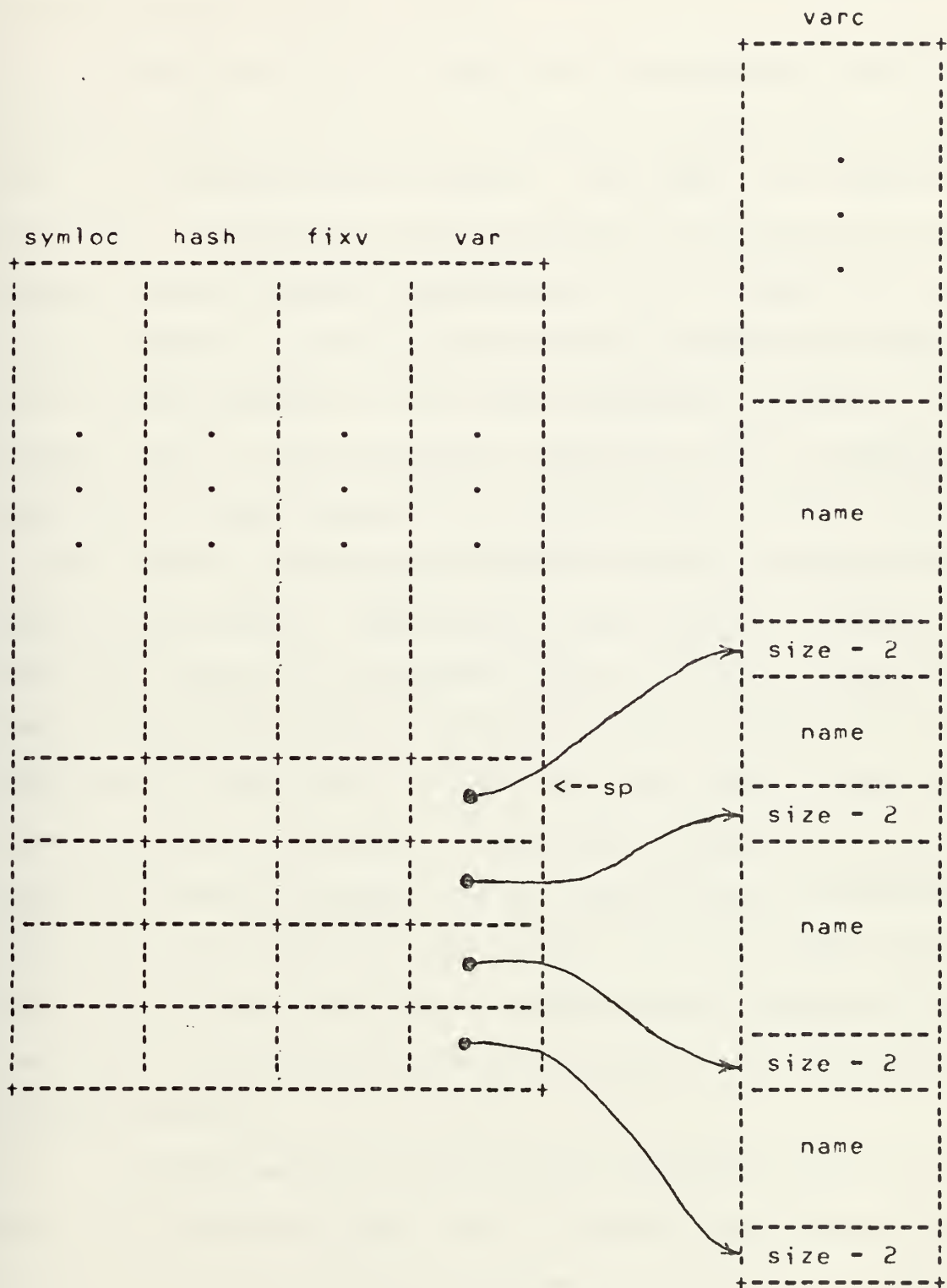
Figure 9.   Scanning and parsing stacks

reported to the parser along with the current value of "sp,"
as discussed in Section IV.B.5.

. The actions for an identifier are somewhat more com-
plicated, since the identifier may be a reserved word, macro
call, or programmer-defined word. In order to determine
which case is applicable, the identifier is looked up in the
symbol table by finding its address in the element of the
vector "hentry" given by the hash code computed by "getto-
ken." If the address is other than the zeroth element of the
symbol table, the printname stored in "varc" is compared
with the printname stored in the table entry. If the names
do not match, the value of "hcoll" is used as the next ad-
dress in the search. This process continues until either a
match is found or the current value of "hcoll" is the ad-
dress of the zeroth element of the symbol table. If an
entry for the identifier is located in the symbol table, the
"type" field is examined to determine whether it is a
reserved word or a macro. In the former case, the reserved
word number is returned to the parser. In the latter case,
the scanner is set up to begin reading input characters from
the "mdef" field of the symbol table entry, and "gettoken"
is called again.

If the identifier is neither a reserved word nor a
macro, information about it is "pushed" onto the parsing
stacks in the manner discussed above for strings. In addi-
tion to the information stored in "var," the address of the
symbol table entry is stored in "symloc," and the hashcode

59

is stored in "hash" (see Figure 9). The "fixv" stack is used to hold other types of information during the parsing process. The fact that an identifier was scanned is reported to the parser along with the current value of "sp."

If the symbol table search is unsuccessful, an entry must be made using the routines in file "m.sym.c." The entry is made immediately following the most recent previous entry. The "hcoll" field of the new entry is set to the value of "hentry[hashcode]", and the value of "hentry[hashcode]" is changed to the address of the new entry. It is assumed at this time that both parts of the "length" field will be required. If it is discovered (during the parsing of later text) that only the first six bits of the "length" field are needed, the "compress" routine (file "m.sym.c") must be called to remove the extra byte in order to save space in the table.

The parser itself uses tables generated from the grammar (file "m.gram") by YACC in order to perform the translation from the source language to the intermediate language (see Chapter V). It does this by shifting tokens onto a set of parsing stacks hidden from the compiler designer. When the tokens match one of the rules of the grammar, a reduction is made by replacing the tokens on the stack with the symbol on the left side of the rule (or production). The methods used for detecting and recovering from errors in the input and the techniques for generating the intermediate language code are discussed in the next two sections.

60

4. Error Recovery

In the discussion of the scanner in Section IV.B.3 it was assumed that the input stream constituted a valid PL/M program. Unfortunately, this is not always the case, especially in the early stages of program development. In addition to the other tasks which a scanner must perform, therefore, it must be able to detect errors and report them to the programmer. One measure of a good compiler is its ability to accurately report all program errors.

Debugging of a large program would be greatly inhibited if the compilation terminated after the detection of a single error. Thus it is desirable for the scanner to have error recovery mechanisms which enable it to continue processing after detecting and reporting an error. The error handling and recovery techniques included in the YACC version of the PL/M compiler are discussed in this section.

There are three basic kinds of errors which may appear in a program--logic, syntactic, and semantic. Logic errors are errors in the programmer's thought processes which cause him to write statements which do something other than what he intended. For example, he might write an expression incorrectly or use the wrong indexing variable when working with a vector. It is impossible for a compiler to detect errors of this type unless they also result in syntactic or semantic errors.

Syntactic errors result from the violation of the grammatical rules of the language. The rules for a

61

programming language like PL/M are given in terms of a series of productions, as in file "m.gram" in Appendix B. One of the main advantages of using a parser derived from such a grammar is that it immediately detects and reports syntactic errors.

Semantic errors are errors which do not violate the rules of the language but which do not have any meaning (or have an incorrect meaning) in the language. It is easy to write nonsense sentences in English which are grammatically correct. An example of a semantic error in a programming language is the use of a variable before it is declared. Some languages allow this, but in the current YACC version of the PL/M compiler this is not allowed, since proper symbol table entries are made only for declaration statements.

At this point it should be helpful to look at an example. Figure 10 lists the sample PL/M program of Figure 1 with several errors intentionally introduced. When this program was run through the compiler the output was as shown in Figure 11. It should be noted that there are two basic types of errors identified in the output. Syntactic errors are identified by the term "syntax error," while semantic errors are identified by the term "compile error."

In order to allow the parser to continue scanning the input after a syntactic error is encountered, YACC allows an "error" production to be included in the grammar. Production 18 in file "m.gram" in Appendix B is the error production used for the PL/M compiler. In this production

```
1.      2048:   /* is the origin of this program */
2.declare tto literally '2', cr literally '15q',
3.         lf literally '0ah',
4.         true literally '1', false literally '0';
5.
6.squareroot: procedure(x byte;
7.     declare (x,y,z) address;
8.     y =x; z = shr(x+1,1);
9.         do while y <> z;
10.        y = z; z = shr(x/y + y + 1, 1);
11.        end;
12.     return y
13.     end squareroot;
14.
15.print$char: proceoure(char);
16.     declare bit$cell literally '91',
17.         (char,i) byte;
18.     output (tto) = 0;
19.     call time (bit$cell);
20.         do i = o to 7;
21.         output(tto) = char; /* data pulses */
22.         char = ror(char,1);
23.         call time(bit$cell;
24.         end;
25.     output(tto) = 1;
26.     call time (bit$cell + bit$cell);
27.     /* automatic return is generated */
28.     end print$char;
29.
30.print$string: procedure(name,length);
31.     declare name address,
32.         (length,i,char based name) byte;
33.         do i = 0 to lengtn - 1
34.         call print$char(char(i);
35.         end;
36.     end print$string;
37.
```

Figure 10. PL/M square root
program with errors
(continued on next page)

```
38.print$number: procedure(number,base,chars,zero$suppress);
39.    declare number address,
40.            (base,chars,zero$suppress,i,j) byte;
41.    declare temp (16) byte;
42.    if chars > last(temp) then chars = last(temp);
43.        do i = 1 to chars;
44.        j = number mod base + '0';
45.        if j > '9' then j = j + 7;
46.        if zero$suppress amd 1 <> 1 and number = 0 then
47.            j = ' ';
48.        temp(length(temp)-i) = j;
49.        number = number / base;
50.        end;
51.    call print$string(.temp+length(temp)-chars, chars);
52.    end print$number;
53.
54.declare i address,
55.    crlf literally 'cr,lf',
56.    heading data (crlf,lf,lf,
57.    '                       table of square roots',
58.    crlf,lf,
59.    ' value   root value  root value   root value   root',
60.    ' value   root',
61.    crlf,lf);
62.
63.    /* silence tty and print computed values */
64.    output(tto) = 1;
65.    do i = 1 to 1000;
66.    if i mod 5 = 1 then
67.        do; if i mod 250 = 1 then
68.            call print$string(.heading,length(heading));
69.        else
70.            call printstring(.(cr,lf,2);
71.        end;
72.    call print$number(i,10,6,true /* true suppresses
73.                                    leading zeroes */);
74.    call print$number(square$root(i),10,6, true);
75.    end;
76.
77.declare monitor$uses (10) byte;
78.eof
```

Figure 10 (continued). PL/M
square root program with errors

```
syntax error, line 6, on input: byte

syntax error, line 13, on input: end

compile error, line 20 : variable undeclared

compile error, line 20 : identifier cannot be a variable

syntax error, line 23, on input: ;

syntax error, line 34, on input: call

compile error, line 35 : identifier required

syntax error, line 36, on input: end

compile error, line 46 : variable undeclared

syntax error, line 46, on input: identifier

syntax error, line 70, on input: ;
```

Figure 11. Compiler output
for program of Figure 10

"error" is a reserved terminal symbol name, and it causes a state to be included in the parser which will be entered any time an invalid symbol is scanned.

When an error is seen, the currently active states are popped, one by one, until a state is reached which has a shift on error. This shift is then done, and the reduction performed. The user may specify an action, to do things such as position the input string and repair the symbol table. After this reduction is done, a flag is set, and the parser remains in error state until three input symbols have been successfully shifted. If an error takes place when the parser is still in error state, the input symbol is discarded and no new message is produced. [30, p.13]

The reason for discarding input symbols if an error occurs while the parser is still in error state is to prevent a simple syntactic error from causing an inordinate number of misleading messages to be generated. Of course, if there are any actual errors in the text while the parser is in the error state they will be ignored. For example, in Figure 11 it can be seen that the parser discovered an error at the beginning of line 34 when it encountered the symbol "call" without scanning a semicolon. Figure 10 shows that there is a missing parenthesis at the end of line 34, and this was not detected by the parser, since it was still in error state. This is not a serious problem, since the parser would detect this error on the second compilation attempt, after the errors detected on the first try were corrected.

The error production used in this compiler causes the parser to scan until finding a semicolon before attempting to continue parsing. This was found to be an effective, although simple, error handling technique. The actions

66

which must be taken to allow parsing to continue without overflowing the various stacks and tables can be seen by examining the listings in Appendix B. Since PL/M is a statement oriented language rather than a card oriented language (such as FORTRAN) and statements are usually relatively short, most errors will be detected by this scheme. In future work. it might prove beneficial to explore additional schemes, such as scanning to a comma or a close parenthesis.

The actions required for detecting and reporting semantic errors are not as easy to specify as are those for syntactic errors, since they are scattered throughout the grammar. For each production in the grammar the compiler designer must consider the meaning of any actions which are to be taken and what circumstances will will cause the actions to be incorrect. For example, the discussion in Section IV.B.2 points out that a procedure may have no more than 63 arguments. Thus the actions associated with the parsing of a "parameter list" (production 42 in file "m.gram") must include a check for the number of arguments. Since it is very difficult to check all possible error conditions of this type, semantic errors are much more difficult to detect effectively than are syntactic errors. In Figure 11, for example, it can be seen that the error on line 20 ("o" in place of "0") causes a redundant error message to be generated. Improvement of the semantic error detection and reporting mechanisms in this compiler would be a worthwhile undertaking for future work.

5.   Semantics and Code Emitting

The method for converting semantic actions, provided by the compiler designer, into a C language program is discussed briefly in Section IV.B.1. In order for the action statements to communicate with the parser, a special notation using "$" variables is employed by YACC. An example of this notation may be seen by examining the action statements associated with production 76 in file "m.gram" (Appendix B). Each symbol on the right-hand side of a production (to the right of the colon) corresponds to a pseudo-variable, the name of which is composed of a dollar sign followed by a digit indicating the relative position of the symbol in the production. Thus "identifier" has the corresponding pseudo-variable "$1" in production 76. There is always one and only one symbol on the left-hand side of a production, and it has the corresponding pseudo-variable "$$" associated with it. The "$" notation is a convenience for the compiler designer, and all pseudo-variables are converted by YACC into actual C language variables before compilation.

In Section IV.B.3 it is stated that the current value of "sp" is passed to the parser when an identifier (other than a reserved word or macro) is scanned. Any such information passed by "yylex" to the parser may be accessed in the action statements by referring to the appropriate "$" variable. Thus, in production 76, the value of "$1" is the value of "sp" received from "yylex." This value is first passed as an argument to the procedure "symcheck" (file

68

"m.act.c"), which checks to see if the variable has been previously declared. Since production 76 is only applied during the parsing of declaration statements, this constitutes a check for a semantic error--the redeclaration of a variable within the same block of the PL/M program. The next three statements are executed if this is the first time the variable is being declared in the current block. First, the value of "fixv[sp]" is set to zero to indicate that this is not a based variable. The next statement ("$$ = sytop") is used to communicate to the next production (possibly 72) the location at which the symbol table entry for this variable begins. The third statement calls the "enter" routine (file "m.sym.c") to actually make the entry in the symbol table. Whether or not an entry is made in the symbol table, the final statement is executed to clear the information associated with this variable from the user-controlled stacks. The parser then makes the reduction indicated by production 76 and stores the information associated with "$$" in one of its parse stacks.

An example of a production which causes intermediate language code to be emitted can be seen in the action statement associated with production 96. In this statement, "$2" refers to a value received from a previously applied production (one of the set 97-102). The "emit" routine (file "m.act.c") is called with two arguments, the first giving the prefix ("OPR") and the second giving the operator determined by the value of "$2" (see Appendix A). The "emit"

routine writes the information onto a disk file which may be used by later stages of the compiler to generate machine code.

It should be noted that the actions associated with production 79 also write information to a disk file using the "putw" routine provide by the C language library. This second output file is used to store all "initial" values declared in the PL/M program being compiled.

One final point can be made by considering the action statements associated with production 33. Productions such as this are connected with the flow of control statements in PL/M, and they cause compiler-generated labels to be produced in order to effect proper branching. Since these labels are often not generated in the same sequence in which they must appear in the intermediate language code, there has to be a mechanism for storing them until they are emitted. As in the case of production 33, labels are generated by incrementing the variable "nsym." Code for a conditional branch is emitted at this point, but the label must be saved until the remainder of the code around which the branch occurs has been generated. This is done by calling the "spush" procedure (file "m.act.c"), which pushes the label onto the "cstack." In order to save space in the compiler, the "cstack" is actually not a separate stack but rather an area at the top of the space allocated to the symbol table.

There are obviously many more details concerning the performance of this compiler than can be presented here.

The YACC reference document [30] should be consulted for a
more complete discussion of the capabilities of YACC, and
the program listings in Appendix B should be studied in or-
der to determine how these capabilities were applied to the
PL/M compiler.

V. THE INTERMEDIATE LANGUAGE

A. FUNCTION

A very important concept in the design of a compiler for user-definable architectures is that of the intermediate language. The compiler model shown in Figure 4 assumes that the source program will be translated to an intermediate form, although it is certainly possible to design a compiler which translates directly to machine code. In fact many compilers have been designed in the latter way, but they lack transportability and are not able to easily take advantage of the more advanced optimization techniques.

The idea of using an intermediate language dates back at least as far as 1958 when there was a discussion of the need for a universal computer-oriented language (UNCOL) in some of the early issues of the Communications of the ACM [12,54,55]. The intent was to have the UNCOL serve as an intermediary between high-level languages and machine languages. This would allow a compiler writer to concentrate on translating from his high-level language to the UNCOL without worrying about the machine code considerations. It would also allow a system programmer for a given machine to write a generator program which produced the best machine code, independent of the high-level language. Whenever a new machine was obtained at a computer installation

only the program which translated from UNCOL to machine code would have to be changed in order to continue using the high-level languages which had been used previously. Old programs could be recompiled without changing the source language. Similarly, whenever a new language was designed it would be necessary only to write a translator which could convert programs written in the new language to UNCOL programs. The new language would then be available to users of any computer for which an UNCOL-machine code translator had been written.

No such universal language has ever been developed, but the concept of an intermediate language has been used by many software designers in writing compilers which could generate code for computers with different architectures and instruction sets (e.g., the PL/M compilers available from Intel for the 8008 and 8080 microprocessors). The fact that an intermediate language is useful in compiling for user-definable architectures is verified by the use of such a mechanism by those who are trying to design high-level languages for microprogrammable machines [18,47].

The main function of the intermediate language in the PL/M compiler is to serve as an information transmission medium between pass 1 and succeeding passes. In this role it is complemented by the symbol table (Section IV.B.2) and the initial value file (Section IV.B.3). The symbol table transmits the names and other attributes of the symbols used in the high-level program, the initial value file transmits

73

the initial values of variables which are to be initialized, and the intermediate language carries information about the actual program steps required by the algorithm. Other information may be contained in the intermediate language code, e.g., the line number markers which are helpful in providing good diagnostics and information to aid the programmer but are not really needed for the code generation process.

Besides its use in transmitting information the intermediate language may have an important role in the process of debugging and simulating the actions of programs translated by the compiler. As explained in Section V.B below, the intermediate language code for the PL/M compiler can be considered to be the "machine code" of a mythical stack machine. It would not be difficult to write an interpreter which could read this code and simulate the actions of the mythical machine in order to help the programmer debug his high-level language program. Broca and Merwin [9] have devoted a paper to this topic, and Reigel and Lawson [48] have indicated that this could provide an important facility.

B. THE POLISH REPRESENTATION

The intermediate language code for the PL/M compiler is based upon postfix Polish notation [3,4,22,42]. The reason for using this type of intermediate language is demonstrated in Figure 12, which shows how an expression written in infix (algebraic) notation would be translated. The bottom-up

74

r = a * b + c * d;

(a)

a b * c d * + r ←

(b)

```
        +,r
        / \
       /   \
      *     *
     / \   / \
    /   \ /   \
   a    b c    d
```
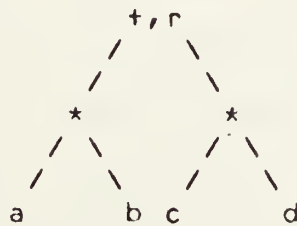
(c)

```
VAL      a
VAL      b
OPR      MUL
VAL      c
VAL      d
OPR      MUL
OPR      ADD
ADR      r
OPR      STD
```

(d)

Figure 12. Example of Polish intermediate language code
(a) infix expression, (b) equivalent postfix expression,
(c) tree representation, (d) intermediate language code

parsing process for the infix expression in Figure 12(a) can be visualized in the tree structure of Figure 12(c), which is a two-dimensional representation of the postfix expression in Figure 12(b). Thus the intermediate language code of Figure 12(d) results naturally from the parsing of the infix expression. Noteworthy is the one-to-one correspondence between symbols in Figure 12(b) and lines of code in Figure 12(d).

This type of intermediate language representation is usually discussed in texts on compiler theory, but the presentations usually do not provide many details about the methods used for an entire practical program. Often the discussion is limited to the type of information presented in Figure 12, but operators other than the simple arithmetic type are required in a language designed to represent real programs. For example, operators are required for branching, subscript calculation, and stack manipulation. The complete list of intermediate language prefixes and operators used in the PL/M compiler, along with their meanings, is given in Appendix A.

In order to provide an example of how the intermediate language is used to represent a program, Figure 13 presents part of the code generated by pass 1 of the PL/M compiler for the square root program of Figure 1. The symbol table for this program can be found in Figure 8. Noticeable in this figure are the expansion factor and the loss of understandability apparent in going from the high-level language

----

```
            LIN     1
            LIT         2048    8   0
            OPR     ORG
            LIN     2
            LIN     3
            LIN     4
            LIN     5
            LIN     6
            OPR     ENP
            VAL     S25
            OPR     TRA
            DEF     S23
            LIN     7
            LIN     8
            VAL     S24
            ADR     S26
            OPR     STD
            VAL     S24
            LIT             1   0   1
            OPR     ADD
            OPR     ARG
            LIT             1   0   1
            OPR     ARG
            VAL     S9
            OPR     BIF
            ADR     S27
            OPR     STD
            LIN     9
            DEF     S28
            VAL     S26
            VAL     S27
            OPR     NEQ
            VAL     S29
            OPR     TRC
            LIN     10
            VAL     S27
            ADR     S26
            OPR     STD
            VAL     S24
            VAL     S26
            OPR     DIV
            VAL     S26
            OPR     ADD
            LIT             1   0   1
            OPR     ADD
            OPR     ARG
```

Figure 13. Intermediate language code for the
program of Figure 1 and symbol table of Figure 8
(continued on following pages)

77

```
                    LIT              1    0    1
                    OPR      ARG
                    VAL      S9
                    OPR      BIF
                    ADR      S27
                    OPR       STD
                    LIN       11
                    VAL      S28
                    OPR      TRA
                    DEF      S29
                    LIN       12
                    VAL      S26
                    OPR      RET
                    LIN       13
                    OPR      END
                    OPR      DRT
                    DEF      S25
                          .
                          .
                          .
code for lines 14 - 53 omitted for brevity
                          .
                          .
                    LIN       54
                    LIN       55
                    LIN       56
                    VAL      S59
                    OPR      TRA
                    OPR      DAT
                    DEF      S60
                    LIT             13    0    D
                    LIT             10    0    A
                    LIT             10    0    A
                    LIT             10    0    A
                    LIN       57
                    LIT              9    0    9
                    LIT              9    0    9
                    LIT              9    0    9
                    LIT              9    0    9
                    LIT              9    0    9
                    LIT              9    0    9
                    LIT             32    0   20
                    LIT            116    0   74    t
                    LIT             97    0   61    a
                    LIT             98    0   62    b
                    LIT            108    0   6C    l
                    LIT            101    0   65    e
                    LIT             32    0   20
                    LIT            111    0   6F    o
```

Figure 13.   (continued)

```
LIT          102    0 66     f
LIT           32    0 20
LIT          115    0 73     s
LIT          113    0 71     q
LIT          117    0 75     u
LIT           97    0 61     a
LIT          114    0 72     r
LIT          101    0 65     e
LIT           32    0 20
LIT          114    0 72     r
LIT          111    0 6F     o
LIT          111    0 6F     o
LIT          116    0 74     t
LIT          115    0 73     s
LIN    58
LIT           13    0 D
LIT           10    0 A
LIT           10    0 A
LIN    59
LIT           32    0 20
LIT          118    0 76     v
LIT           97    0 61     a
LIT          108    0 6C     l
LIT          117    0 75     u
LIT          101    0 65     e
LIT           32    0 20
LIT           32    0 20
LIT          114    0 72     r
LIT          111    0 6F     o
LIT          111    0 6F     o
LIT          116    0 74     t
LIT           32    0 20
LIT          118    0 76     v
LIT           97    0 61     a
LIT          108    0 6C     l
LIT          117    0 75     u
LIT          101    0 65     e
LIT           32    0 20
LIT           32    0 20
LIT          114    0 72     r
LIT          111    0 6F     o
LIT          111    0 6F     o
LIT          116    0 74     t
LIT           32    0 20
LIT          118    0 76     v
LIT           97    0 61     a
LIT          108    0 6C     l
LIT          117    0 75     u
```

Figure 13.   (continued)

```
LIT          101    0 65    e
LIT           32    0 20
LIT           32    0 20
LIT          114    0 72    r
LIT          111    0 6F    o
LIT          111    0 6F    o
LIT          116    0 74    t
LIT           32    0 20
LIT          118    0 76    v
LIT           97    0 61    a
LIT          108    0 6C    l
LIT          117    0 75    u
LIT          101    0 65    e
LIT           32    0 20
LIT           32    0 20
LIT          114    0 72    r
LIT          111    0 6F    o
LIT          111    0 6F    o
LIT          116    0 74    t
LIN    60
LIT           32    0 20
LIT          118    0 76    v
LIT           97    0 61    a
LIT          108    0 6C    l
LIT          117    0 75    u
LIT          101    0 65    e
LIT           32    0 20
LIT           32    0 20
LIT          114    0 72    r
LIT          111    0 6F    o
LIT          111    0 6F    o
LIT          116    0 74    t
LIN    61
LIT           13    0  D
LIT           10    0  A
LIT           10    0  A
OPR    DAT
DEF    S59
LIN    62
LIN    63
LIN    64
LIT            2    0  2
OPR    ARG
LIT            1    0  1
OPR    XCH
OPR    STD
LIN    65
LIT            1    0  1
```

Figure 13.   (continued)

```
            ADR     S58
            OPR     STD
            DEF     S61
            VAL     S58
            LIT         1000    3 E8
            OPR     LEQ
            VAL     S62
            OPR     TRC
            LIN     66
            VAL     S58
            LIT             5   0   5
            OPR     REM
            LIT             1   0   1
            LIN     67
            OPR     EQL
            VAL     S63
            OPR     TRC
            OPR     ENB
            VAL     S58
            LIT           250   0 FA
            OPR     REM
            LIT             1   0   1
            LIN     68
            OPR     EQL
            VAL     S64
            OPR     TRC
            ADR     S60
            OPR     CVA
            OPR     ARG
            VAL     S60
            OPR     ARG
            VAL     S17
            OPR     BIF
            OPR     ARG
            VAL     S36
            OPR     PRO
            LIN     69
            LIN     70
            VAL     S65
            OPR     TRA
            DEF     S64
            VAL     S66
            OPR     TRA
            OPR     DAT
            DEF     S0
            LIT            13   0   D
            LIT            10   0   A
            OPR     DAT
```

Figure 13.   (continued)
```

81
```

```
DEF     S66
OPR      ARG
LIT             2    0    2
OPR      ARG
VAL     S36
OPR      PRO
DEF     S65
LIN      71
OPR      END
LIN      72
DEF     S63
VAL     S58
OPR      ARG
LIT            10    0    A
OPR      ARG
LIT             6    0    6
OPR      ARG
LIT             1    0    1
LIN      73
OPR      ARG
VAL     S44
OPR      PRO
LIN      74
VAL     S58
OPR      ARG
VAL     S23
OPR      PRO
OPR      ARG
LIT            10    0    A
OPR      ARG
LIT             6    0    6
OPR      ARG
LIT             1    0    1
OPR      ARG
VAL     S44
OPR      PRO
LIN      75
VAL     S58
OPR      INC
ADR     S58
OPR      STO
VAL     S61
OPR      TRA
DEF     S62
LIN      76
LIN      77
LIN      78
LIN      79
```

Figure 13.   (continued)

to the intermediate language. Of course the computer, through the actions of the compiler, is much better equipped to cope with this than the human programmer trying to write this program in assembly language.

The postfix Polish code is often referred to as zero-address code since the operator instructions are intended to manipulate values on the top of a push-down stack and thus do not contain an address field. The method generally used to generate machine code from this zero-address code is to simulate a mythical stack machine in the compiling process. This "meta-execution" stack of course does not usually contain values, since most of the variables in a program have values assigned at execution time rather than at compile time, but rather it contains information about the program symbols. This type of code generator is fairly simple to implement, especially if optimization is not too important, and should be fairly easy to adapt to a table-driven scheme (see Chapter VIII).

It should be noted that each "instruction" in the intermediate language consists of two parts, a prefix and an operator or operand. The prefix indicates the type of the instruction, while the second part is an operator (e.g., MUL, ADD, TRA) for an OPR prefix or an operand for other prefixes. The LIN instruction is used to transmit line numbers from the source program, and the DEF instructions define labels in the intermediate code for purposes of branching. The VAL and ADR instructions place values and

83

addresses, respectively, on the stack, while the LIT in-
struction places literal (numeric or immediate data) values
on the stack. The second field of the LIT instruction is
presented in four columns in Figure 13. The first column
gives the decimal value of the 16-bit literal (range:
0-65535), and the second and third columns give the hexade-
cimal values of the high and low order bytes, respectively.
The fourth column indicates the two ASCII characters (if
printable) represented by the value.

C. OTHER REPRESENTATIONS

The reverse Polish form is not the only one used for
intermediate representation of programs. The two most com-
monly used alternatives are triples and quadruples, the
former being equivalent to two-address code and the latter
to three-address code. Using this terminology, the Polish
code could be referred to as "singles."

Triples are more clearly representative of the tree
structure of a program than zero-address code, since each
triple has the form

        (operator, operand1, operand2),
and the triples are linked by pointers to show the flow of
control (either operand may actually be a pointer to another
triple whose result is used in the current triple). One
difficulty with this method is that it requires more memory
to represent the program than the Polish method, but Gries
[22] presents a method for modifying the implementation to
reduce the memory requirement.

84

Quadruples have a "result" field in addition to the three fields of the triple and take the form

(operator, operand1, operand2, result),

where the fourth field is either a temporary variable generated by the compiler (in the case of a subexpression of an arithmetic expression) or a program variable (e.g., the variable on the left-hand side of an assignment). Some quadruples (and triples also) will require only the operator and one operand (e.g., a branch instruction), while others will require all fields but "operand2" (e.g., for the unary minus operator: $y = -x ==> (-,x,,y))$. The difficulties with quadruples are a greater memory requirement than for the Polish form and the large number of temporary variables which would be generated for any significant program.

Many compiler designers prefer triples or quaduples to Polish code [18,50,41], because these forms are claimed to be easier to manipulate for optimization purposes. This is a point which deserves further investigation; however, it should be noted that powerful optimization techniques have been successfully applied to programs represented by a Polish intermediate language [13].

VI. DIGITAL SYSTEM DESCRIPTION

Once the source program has been translated into an intermediate form and its descriptive information has been preserved in tables, the job of converting this information into control code begins. The remaining stages of the compiler require, in addition to the information transmitted from pass 1, detailed knowledge of the architecture and instruction set of the hardware in order to accomplish the code generation task. Ordinarily this other information would be included in the remaining stages at the time the compiler was designed, but this cannot be done if the architecture is unknown prior to compile time. Thus this chapter is concerned with the types of information required and the problem of describing this information for varying architectures.

Many languages have been developed for describing digital systems, and two excellent surveys of these languages have been published [6,25]. Because most of these languages were developed by individuals or small groups of individuals working on specific problems they have shortcomings which do not allow universal application. For this reason the Conference on Digital Hardware Languages, a special continuing conference of experts in the computer hardware description field, has been formed in an attempt to define a language which can become a standard for the industry [37].

Although the main purpose of such languages is to serve as aids in designing and simulating digital systems, it should be obvious that they could also be used to describe a system as part of the compilation process.

Since there are several levels of detail which may be used to describe a digital system the first problem is to choose the most appropriate one. Bell and Newell [7] have defined a hierarchy of five levels for description of computer systems: the circuit level, the switching circuit level, the register transfer (RT) level, the programming level, and the PMS (processor, memory, switch) level. .The circuit level is the lowest level and is well established, with a notation and set of conventions which have become standardized over many years of electrical engineering practice. During the relatively few years that digital electronics has been in existence the switching circuit level has also become well established, allowing designers to avoid much of the detail necessary in describing their systems at the circuit level. Thus digital circuits are designed with gates and delays rather than transistors, diodes, and other components of the circuit level. At the other end of the scale, the PMS level (although the specific term was coined by Bell and Newell) has also been in use for some time, since this is the level used to describe the gross properties of computer systems. The programming level has also become well developed, since most digital systems have been the kind which require a program in order to perform useful

87

functions. The RT level, which is the one that seems most natural for conveying the structure of digital systems and interfacing between the circuit levels and the programming level, has been recognized as a level since the 1950's but has only recently been the subject of serious efforts aimed at formalization [6].

During the brief history of the computer industry computers have evolved from huge pieces of hardware with very limited capabilities (by today's standards) to very compact units with very broad, powerful capabilities. For all of this change, though, the architectures of computer systems still closely adhere to the concepts originally proposed by von Neumann [7, ch.4]. Even the development of minicomputers and microcomputers has not changed this fact, since most of the same features which were successful on larger computers have been carried over into these smaller systems. In fact, the increased competition in the computer industry which has been caused by the acceptance of these new types of computers will probably have the effect of "weeding out" features which are not well conceived or introduced merely for uniqueness and of more or less standardizing features which prove useful across a wide range of applications.

No attempt will be made here to describe all of the variations in architecture which have evolved over the years, since a comprehensive survey has been presented by Bell and Newell [7]. Suffice it to say that, while there are many differences among the various types of systems at

the switching circuit level, there are many similarities at the RT level. The features which distinguish one system from another at this level can be grouped according to a few system characteristics.

A. BASIC CHARACTERISTICS

The two main PMS level building blocks in a conventional system are the central processing unit (CPU) and the memory. In most systems the majority of the instructions are devoted to transferring data between these two units. In order to represent these ideas at the RT level Barbacci [6] refers to the basic components as "operators" and "carriers."

> Operators are entities that produce information by transformation of bit patterns to which meaning has been assigned. These bit patterns reside in carriers, which are the entities used in storing and transmitting the information to and from the operators. [6, p.139]

The operators can be seen to represent the functions performed by the CPU in the classical digital system, and the carriers are hardware memory elements with various degrees of latency. Wires and busses can be considered short term memories, while registers and core memories carry information for longer intervals of time.

The basic characteristics of a digital system can thus be described by defining the various carriers and operators of which it is constructed. Barbacci considers the register to be the basic unit in defining carriers, and its descriptors are its name, dimensions, and size of alphabet (typically 2, as in binary systems). All types of memory in a system can be constructed from registers by forming

89

subregisters, compound registers, and arrays. An example of subregisters in a typical system is the partition of the instruction register into an operation code field and one or more operand fields. The Intel 8080 microprocessor provides examples of compound registers; e.g., the H and L registers are considered as separate registers in several instructions, but when concatenated they form the address register. The primary memory of a typical computer can be thought of as an array of registers.

The two most primitive kinds of operators in a digital system are the ones usually represented in a high-level language: logical (negate, inclusive or, exclusive or, and, equivalence) and arithmetic (addition, subtraction, mutiplication, division). Other operators needed in describing the system include vector operators for manipulating registers (shift, rotate), transfers, concatenation, and special operators (counting, exchanging, etc.).

It is necessary but not sufficient for a compiler to have information about the operators and carriers of a computer system. Since the primary purpose of the compiler is to generate control code for the computer, information describing the instruction set is also required. In essence the compiler performs a mapping from intermediate language code to machine code, and it is necessary to provide sufficiently detailed information to carry out this mapping. The level of detail will be much greater if the intermediate language is to be translated into microcode than if it is to be translated into a conventional machine language. The bit

90

patterns of all of the machine instructions are required in order to produce the actual machine code, and the mnemonics of the instructions (in an assembly language type of format) may be required in order to produce a version of the machine code which can easily be read by the programmers who will be trying to debug the programs produced. Special types of instructions (subroutine jumps and returns, interrupt producing and handling, input/output) must be accounted for, and any side effects of instructions (such as the setting of condition codes) must be described. Methods of addressing will have an effect on the code produced and will also have to be described. For some machines it is necessary to transfer operands from main memory to registers in order to operate on them, while other machines have instructions which operate on operands directly in main memory. Many machines have both types of instructions. Indirect addressing, indexing, and stack manipulation are important features which also must be described.

The amount of detail provided about the instruction set and the physical characteristics of the machine has a direct bearing on the capability of the compiler to produce "good" machine code. If sufficient information is available machine-dependent optimizations can be performed on the code as it is being generated, as discussed in Section VII.A.1.

B. MICROPROGRAMMING AND MODULARITY

Two basic classes of systems can be distinguished, the first being the conventional or classical type,

91

characterized by a fixed or hardwired instruction set. Microprogrammable systems, in which the instruction set (in the usual sense of the term) is able to be changed by altering a memory, form the second class. Included in the latter class are modular systems, which in addition to having a variable instruction set have a variable machine organization. Both classes of systems require the basic types of information discussed above to be transmitted to the compiler. The additional types of information required by the compiler for microprogrammable systems are discussed below.

Microprogrammable systems have been growing rapidly in use in the last few years, but for all of the special attention which has been devoted to it, microprogramming is not significantly different from "regular" programming. Reigel and Lawson have defined microgramming as "... a technique for implementing the control function of a digital computing system as sequences of control signals that are organized on a word basis and stored in a memory." [48, p.2] There is nothing in this definition which does not apply equally well to a non-microprogrammed computer. Eckhouse has noted that, with respect to microprogrammable hardware, "... all of the machines can be classified as classical, von Neumann in nature with only minor perturbations." [18, p.172]

What microprogramming has done is allow increased flexibility in digital system design by providing the designer with greater access to the hardware. IBM was the first company to successfully apply microgramming when it produced the S/360 family of computers. The designers were able "...

92

to achieve a range of compatible processors offering the same large machine instruction set at many different levels of performance." [8, p.3] It is interesting to note that, in a sense, the common machine code of this series could be considered a machine-independent programming language.

Perhaps the attraction of microprogramming can best be appreciated by considering the distinction which Barbacci makes between architecture and machine organization. He considers the architecture to be the behavioral description of a system; i.e., what the programmer perceives the system to be. On the other hand, the machine organization is "... the particular combination of registers, busses, combinational networks, and control ..." in a system. [6, p.144]

> The architecture influences the machine organization by imposing a set of requirements (a particular instruction set) and the organization, mainly for technological reasons, influences the architecture of the machine. The result is usually that a given computer architecture can be implemented on a set of machine organizations, and a given organization accepts several architectures. [6, p.144]

Thus in a conventional computer system it is necessary to describe only the architecture of the machine. The instruction set serves as a level of abstraction separating the programmer (or compiler) from the machine organization. In a microprogrammed system another instruction set may be defined in order to preserve this abstraction, but if the programmer is to work in a high-level language it may be possible (but not necessarily desirable) to skip this step by allowing the compiler to interface directly with the machine organization.

The major hurdle in microprogramming is the introduction of timing considerations to the list of information required. In describing the basic information required by a compiler, no mention was made in Section VI.A of the timing of instructions. The reason for this is that conventional systems usually operate in a sequential fashion, with the execution of one instruction not beginning until the previous instruction has been executed. Even though many events may be occurring in parallel, this is hidden by the instruction set. One of the values of microprogramming is that it allows the programmer to specify the concurrency of certain events. Unfortunately this additional flexibility is obtained by increasing the amount of detail with which the programmer must cope. This is why instruction sets similar to those of conventional computers are usually defined for microprogrammable systems. For example, the Intel 3000 series of microprocessor chips has been advertised as a microprogrammable microprocessor; however, the series has not yet been exploited to its fullest potential because Intel is still in the process of defining a higher-level instruction set comparable to the typical machine language. More of the considerations involved in working with concurrent systems are presented below in Section VI.C.

Another trend which is occurring in the digital electronics field is the development of modular systems [16,31,56]. The reasons for development of such systems are very similar to the reasons for some of the software

94

engineering concepts (see Section I.B). In fact, modularity
(of programs) is one of the principles of software engineer-
ing. In addition to the variable architecture characteris-
tic of other microprogrammable systems, modular systems have
variable machine organizations. Thus the task of program-
ming these systems is even more difficult.

    If development of different types of components can be
    reduced, and if standardization of modules, test pro-
    cedures, and logistic support can be achieved, the life-
    cycle cost of systems can be greatly reduced. One
    approach to implementation of this idea is to identify a
    level of modularity for components which can have wide
    application in many types of systems. This allows the
    development cost of the modules to be spread over many
    units, while reducing the quantity of components and the
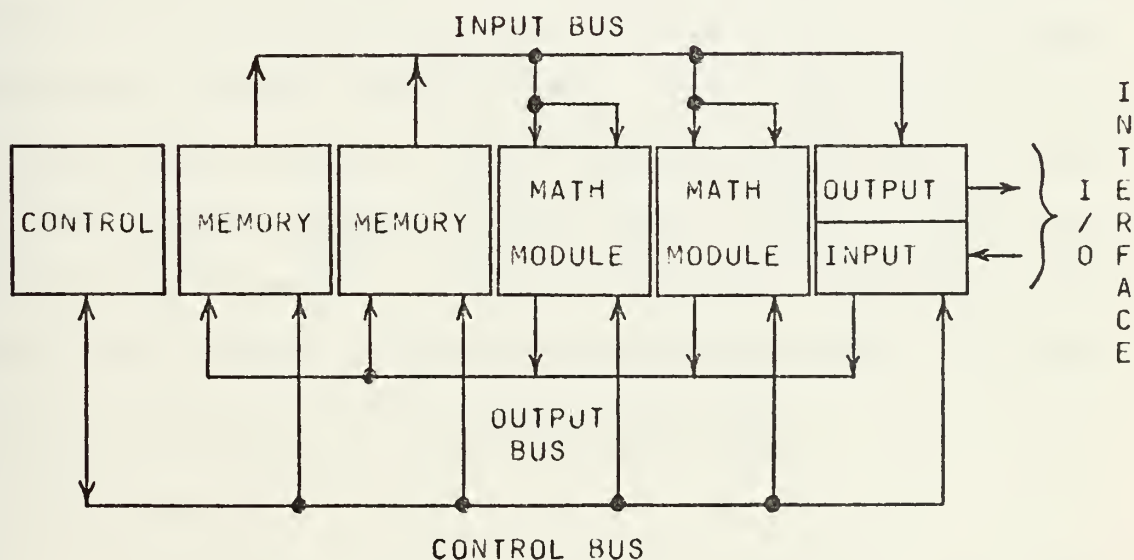   logistics costs. [56, p.3]

Figure 14. Example of a three-bus
modular system using QED modules

95

One of many possible configurations for a modular system is shown in Figure 14 [56, p.19]. The control module in this type of system would probably consist of a read-only memory programmed to initiate all actions required by the system performance specifications. This type of system would be very similar to a microprogrammable computer in its operation and control code structure. Thus the main problem in programming such a system will be to take maximum advantage of parallelism. The designer will also have the problem of selecting the most appropriate types and numbers of modules to use and the problem of choosing the most efficient organization. (i.e., the number of busses and the connections of modules to busses).

An alternative type of modular system would spread the control function among all of the modules rather than consolidating it into a single module. Such a system would not be program controlled but would accomplish its tasks by having the various modules communicate with one another by means of "ready" and "acknowledge" signals. Because such modules would not be as flexible as the type shown in Figure 14, they probably will not be as widely used.

C. PARALLELISM

The rapid growth of the computer industry has been spurred by the ever-increasing speed of computer hardware brought about by continuing advances in the electronic component industries. Vacuum tubes gave way to transistors in the late 1950's and early 1960's, and the latter were

supplanted by integrated circuits in the mid-1960's. These small-scale integrated (SSI) circuits were soon antiquated by medium-scale integration (MSI) technology, and direct gate-level and register-level design (without considering the circuit-level) became possible. Today large-scale integrated (LSI) circuitry is available in large quantities, and whole subsystems can be manufactured on one small chip of silicon. While these tremendous increases in circuit density have played an important part in increasing the speed of digital systems, they have been accompanied by advances in the state-of-the-art in semiconductor manufacture which have allowed much faster switching times to be achieved.

Unfortunately, there are physical limits to the processes which have brought about these vast changes, and the semiconductor industry will soon be nearing them. As a result, increases in computer circuit speed will be coming at a much slower rate than in the past (unless some new technology is discovered which does not depend on the motion and storage of electrons). Thus any further major advances in computer speed will have to rely on increased use of advanced machine organization techniques which take advantage of features such as parallelism and pipelining. Parallelism refers to the concurrent performance of multiple tasks in a system, where a task is "... a self-contained portion of a computation [or some other computer operation] that once initiated can be carried out to its completion

97

without need for additional inputs." [46, p.986] Pipelining is accomplished by dividing a task into many independent subtasks such that a new process can begin the task as soon as the previous one has completed the first subtask. In this way many processes can be performing the same task, each being in a different stage of completion.

Another factor which has resulted in the increased use of parallelism in digital systems is the growing emphasis on modularity and microprogramming discussed in Section VI.B. Modularity in hardware may be looked at from several viewpoints, from small modules such as registers and busses considered in microprogramming to large functional modules as discussed by Tinklepaugh and Eddington [56]. This wide diversity means that there are several levels of parallelism which must be considered, each with its own unique problems and methods of solution [46].

The fact that parallelism is a significant consideration in the design of a digital system is highlighted by the fact that at least one entire book has been devoted to the subject [39]. Several high-level language compilers have been developed or proposed for use with the new generation of array processors, which rely heavily on the use of parallelism at the instruction and arithmetic expression levels [35]. Ramamoorthy, Park, and Lee [46] present a good overview of some of the factors involved in working with parallelism and present several algorithms for taking advantage of it at the arithmetic expression and subexpression levels.

There are two basic ways in which parallelism may be handled in a high-level language--explicitly or implicitly. In the explicit approach there are special instructions included in the language (e.g., FORK and JOIN) by which the programmer may indicate sections of code which may be executed in parallel.

> The explicit approach is advantageous for the recognition and representation of parallelism between blocks of instructions or between instructions, since the analysis of parallelism between tasks at these levels is simple. However, the explicit approach is not advantageous for recognizing and representing parallelism at arithmetic operation (subexpression) or micro-step level, because it is tedious and mistake prone. [46, p.986]

The implicit approach places the burden on the compiler by incorporating two new steps in the compilation process. The first step involves the recognition of parallel processable tasks, and the second involves representation of the information obtained in the first step and allocation of resources in such a manner that maximum advantage is taken of the parallelism. "This approach involves considerable overhead to recognize parallel tasks in a program although it relieves programmers of additional duties." [46 p.986]

Thus, as is usually the case in design work, there are tradeoffs involved in determining whether to use the explicit or the implicit approach to parallelism. At the current state-of-the-art in compiler design it is probably desirable to use some combination of the two. The explicit approach can be used for blocks of instructions (e.g., subroutines), and the implicit approach can be used at the arithmetic expression level to reduce the number of programming errors

which would result from using the explicit approach at this level.

In either approach the compiler can be given the job of allocating the resources for execution of the program. The major problem in this area involves the proper synchronization of the various pieces of hardware. Again there appear to be two possible methods for approaching this problem. One way would require that the description of each module contain information about the maximum time required for the module to perform a given function. Then once the compiler had assigned a task to a particular module it would have to allow the specified amount of time before it could issue an instruction requiring the results of that module to be available. This approach has the disadvantage that it would not allow the hardware to operate at maximum speed, since many functional modules (e.g., multipliers) have a wide variation in speed depending on the input data. The other approach is the one used in modern operating systems for sophisticated computers. In essence this would involve setting up a small operating system which would perform the resource allocation at execution time rather than compile time. Synchronization would be accomplished by sending control signals to the various modules and receiving signals from the modules to indicate task completion. Obviously this method involves a fairly significant amount of overhead in the form of additional memory required to hold the operating system. Thus the programmer would have another

tradeoff to make in determining which method would be most
appropriate for his application.

As in the other sections of this chapter, many ideas
have been presented in this section. No specific recommen-
dations have been made as to which of them may be applicable
to a compiler for user-definable architectures, since the
determination of such recommendations will require a consid-
erable amount of additional research and experimentation.
The intent here has been to exhibit a (not necessarily all-
inclusive) list of some of the things which must be con-
sidered in implementing "pass 2" of such a compiler.

VII. COMPILER OPTIMIZATION

Optimization is a frequently pursued goal in the design of engineering systems, whether they be hardware systems or software systems. The mathematical solution of an optimization problem requires finding the minimum of a cost function (or maximum of a reward function) while satisfying a set of constraints. Unfortunately the equations involved are often nonlinear, making a closed-form solution impossible. Attempts at solution by enumerating all the possibilities are usually not practical for nontrivial problems, because the enumeration expands in a combinatorial manner. (In optimal control theory this problem has sometimes been referred to as the "curse of dimensionality.") Thus, though a large body of theory has been developed to deal with optimization problems, often the only practical solution to a problem involves the use of ad hoc methods. Such has been the case to a large extent in dealing with the problem of code optimization.

Another significant barrier to the application of good optimization techniques is the general difficulty of specifying what constitutes an optimal solution to a given design problem. In fact it has been noted by Aho and Ullman, with respect to the code generated by a compiler,

> ... that there is no algorithmic way to find the shortest or fastest-running program equivalent to a given program. ... Thus the term optimization is a complete misnomer--in

practice we must be content with code improvement. Various code improvement techniques can be employed at various phases of the compilation process.   [3, p.70-71]

It is the purpose of this chapter to discuss the motivation for research in the area of compiler optimization and to examine some of the formal techniques which may prove useful in implementing compilers for firmware design languages.

A.   MOTIVATION

As far back as the early 1950's, when the FORTRAN I compiler was being designed, it was recognized that convenience alone was not enough to persuade programmers to use high-level languages [52].   Unless the compiler could produce machine code which was comparable in efficiency to hand-coded programs there would be a great deal of resistance to the use of high-level languages. In the intervening years computer architectures and instruction sets have increased in complexity, making it even more difficult for a compiler to match a good assembly language programmer.

Three computer hardware trends which have developed over the years are the increase in speed, the increase in main memory size, and the increase in size and power of the instruction set. These trends have had the effect of reducing the need for optimization in compilers, since for many applications the hardware efficiency more than offset the compiler-generated code inefficiency. In recent years there has been yet another trend--the acceptance of minicomputers, and now microcomputers, as components in the design of larger systems. In such applications the cycle is beginning

103

to repeat, since these smaller computers typically have slow execution times, a small amount of memory, and a relatively limited number of instructions. Thus the programs written for these devices must be as efficient as possible in order to minimize the amount of hardware used. Even in sophisticated microprogrammable systems, though, the amount of hardware used may be critical in determining the profitability of a given design. As a consequence, code optimization is becoming increasingly important in order to allow firmware designers to take advantage of all the benefits of high-level language programming.

An example of the kinds of inefficiencies involved is shown in Figures 15-17. Figure 15 shows a PL/M program for performing a simple bubble sort, while Figure 16 shows a hand-coded Intel 8080 assembly language version of the same program [43]. Note that neither of these programs would actually be run by itself but would probably be a procedure in a larger program (in which ARRAY and N would be given values). The purpose here is to examine the code generated for the sorting algorithm without getting involved in the various issues of subroutine linkage.

Figure 17 shows the output (reformatted by the author for ease of comparison) of the Intel 8080 PL/M compiler, version 1.0. Not counting storage space for the variables, the hand-coded version requires 40 bytes of storage, and the compiler version requires 116 bytes--a relative inefficiency of 190 percent. A similar but somewhat larger version of

```
1.    DECLARE ARRAY(256) BYTE,
2.      (N,I,T1,T2,SWITCHED) BYTE;
3.    SWITCHED = 1;
4.      DO WHILE SWITCHED;
5.      SWITCHED = 0;
6.        DO I = 1 TO N - 1;
7.        T1 = ARRAY(I); T2 = ARRAY(I + 1);
8.        IF T1 > T2 THEN
9.            DO;
10.           ARRAY(I+1) = T1;
11.           .   ARRAY(I) = T2;
12.           SWITCHED = 1;
13.           END;
14.       END;
15.     END;
16.   EOF
```

Figure 15.   PL/M bubble sort program

```
1.          MVI     D,1
2.    L1:   MOV     A,D
3.          ADI     0
4.          JZ      L2
5.          MVI     D,0
6.          MVI     H,N.H
7.          MVI     L,N.L
8.          MOV     B,M
9.          MVI     H,ARRAY.H
10.         MVI     L,ARRAY.L
11.   L3:   DCR     B
12.         JZ      L1
13.         MOV     A,M
14.         INR     L
15.         CMP     M
16.         JP      L3
17.         MOV     C,M
18.         MOV     M,A
19.         DCR     L
20.         MOV     M,C
21.         INR     L
22.         MVI     D,1
23.         JMP     L3
24.   L2:   HLT
```

Figure 16.   Hand-coded 8080 assembly language
version of bubble sort program (after Popper [43])

```
 1.        LXI     SP,00F4H          35.        DCR     L
 2.        LXI     H,SWITCHED        36.        SUB     M
 3.        MVI     M,1               37.        JNC     L3
 4. L1:    LXI     H,SWITCHED        38.        DCR     L
 5.        MOV     A,M               39.        MOV     C,M
 6.        RRC                       40.        INR     C
 7.        JNC     L4                41.        MOV     L,C
 8.        MVI     M,0               42.        MVI     H,0
 9.        MVI     L,I.L             43.        LXI     D,ARRAY
10.        MVI     M,1               44.        DAD     D
11. L2:    LXI     H,N               45.        XCHG
12.        MOV     C,M               46.        LXI     H,T1
13.        DCR     C                 47.        MOV     C,M
14.        MOV     A,C               48.        MOV     A,C
15.        INR     L                 49.        STAX    D
16.        SUB     M                 50.        LHLD    I
17.        JC      L1                51.        MVI     H,0
18.        LHLD    1                 52.        LXI     D,ARRAY
19.        MVI     H,0               53.        DAD     D
20.        LXI     D,ARRAY           54.        XCHG
21.        DAD     D                 55.        LXI     H,T2
22.        MOV     A,M               56.        MOV     C,M
23.        LXI     H,T1              57.        MOV     A,C
24.        MOV     M,A               58.        STAX    D
25.        DCR     L                 59.        INR     L
26.        MOV     C,M               60.        MVI     M,1
27.        INR     C                 61. L3:    MVI     L,I.L
28.        MOV     L,C               62.        MOV     C,M
29.        MVI     H,0               63.        INR     C
30.        LXI     D,ARRAY           64.        MOV     M,C
31.        DAD     D                 65.        JNZ     L2
32.        MOV     A,M               66.        JMP     L1
33.        LXI     H,T2              67. L4:    EI
34.        MOV     M,A               68.        HLT
```

Figure 17. Reformatted PL/M compiler output
for bubble sort program of Figure 15

this program, cited by Falk [19], was hand-coded for the Intel 8008, and required 347 bytes of code. The initial version of the 8008 PL/M compiler generated 495 bytes of code for this larger program--a relative inefficiency of 47 percent. A later version of the 8008 PL/M compiler (containing improved optimization techniques) generated 388 bytes of code, yielding a 12 percent relative inefficiency [34].

This tends to confirm the fact that, in most applications, compiler-produced code compares more favorably with assembly code as the size of the program increases. Also the current PL/M compilers use relatively unsophisticated optimization techniques, and further improvements could be obtained with relatively little additional effort.

Comparison of Figures 16 and 17 shows that the bubble sort program brings out two of the most severe problems in compiler code generation--the register allocation problem and the subscript calculation problem. Less significant, but also evident, are the differences in the methods of branching for the loops and the four extra bytes of code generated by the compiler for all programs (to set the stack pointer at the beginning and enable interrupts at the end).

The assembly language version takes advantage of the fact that there are enough index registers available on the 8080 to hold all temporary variables needed in the sort routine. This saves at least three bytes of code (to load the address into the HL register) each time one of these

variables is referenced (unless the address is already in HL from a previous reference).

The greatest saving in the program of Figure 16 results from the use of the HL register as a pointer into the array being sorted. The programmer realized that the elements of the array being referenced at any time were always adjacent to one another, and he stepped through the comparisons and swaps by appropriately incrementing and decrementing the address register. The current compiler is not capable of making this optimization and so recomputes subscripts for each variable reference.

An attempt was made to rewrite the PL/M program to more closely match the structure of the assembly language version (see Figure 18). In the new program the iterative loop was replaced with a WHILE loop, and the swapping process was modified so that it would use only one temporary variable. Unfortunately the savings produced by these changes were offset by the computation of one additional subscript, and the new program generated as much code as the old.

As mentioned in Chapter II, one of the advantages of programming in a high-level language is the ease with which changes can be made. Figure 19 shows the minor PL/M program changes (to the declaration statements) which would be re-quired if the array to be sorted contained more than 256 values and if the values were double-byte (address) rather than single-byte. Two other changes have been indicated in order to make the program technically correct. Of course

```
1.    DECLARE ARRAY(256) BYTE,
2.        (SWITCHED,N,I,TEMP) BYTE;
3.    SWITCHED = 1;
4.        DO WHILE SWITCHED;
5.        SWITCHED = 0;
6.        I = N;
7.            DO WHILE (I := I - 1);
8.            IF ARRAY(I) > ARRAY(I+1) THEN
9.                DO;
10.                TEMP = ARRAY(I);
11.                ARRAY(I) = ARRAY(I+1);
12.                ARRAY(I+1) = TEMP;
13.                SWITCHED = 1;
14.                END;
15.            END;
16.        END;
17.    EOF
```

Figure 18.   PL/M program revised to match
the control structures of the assembly language version

```
1.    DECLARE ARRAY(256) ADDRESS,
2.        (N,I,TEMP) ADDRESS,
3.        SWITCHED BYTE;
4.    SWITCHED = 1;
5.        DO WHILE SWITCHED;
6.        SWITCHED = 0;
7.        I = N - 1;
8.            DO WHILE (I := I - 1) + 1;
9.            IF ARRAY(I) > ARRAY(I+1) THEN
10.                DO;
11.                TEMP = ARRAY(I);
12.                ARRAY(I) = ARRAY(I+1);
13.                ARRAY(I+1) = TEMP;
14.                SWITCHED = 1;
15.                END;
16.            END;
17.        END;
18.    EOF
```

Figure 19.   Modified PL/M program

the changes caused much more code to be generated (197 bytes as opposed to 116) in order to handle the double-byte arithmetic and data transfer operations. The interesting point here is that an assembly language version, if one had been written for this new problem, would certainly be much longer than 40 bytes since there would be insufficient registers available to hold all of the temporary values. Also the compiled version would have a much lower relative inefficiency in relation to such a hand-coded version of the new program. The amount of effort required to change the program would obviously have been many times greater than was the case for the PL/M version.

B. TECHNIQUES

In his excellent compiler optimization survey Schneck [52] has classified optimization techniques into three functional categories based upon the amount of knowledge they require about the object machine. He calls the three categories machine-dependent, architecture-dependent, and architecture-independent. Some of the more important techniques in each category are highlighted below in order to show that many of the inefficiencies usually associated with compiler-generated code can be eliminated if careful attention is paid to optimization.

1. Machine-Dependent

Machine-dependent optimizations are also classified as local optimizations since they are applied to short spans of code during the code generation process rather than prior

110

to code generation as indicated in Figure 4. Thus these techniques require a detailed knowledge of the instruction set of the object machine. For example, if the operation to be performed is an addition and one of the operands is known at compile time to have a value of one, the code generated would be an increment instruction if one were available.

The majority of the optimizations in pass 2 of the Intel PL/M compilers fall into this category. The results of some of the more subtle ones can be seen in Figure 17. It should be noted that the MVI instruction has been used whenever possible to perform data transfers. This instruction requires two bytes of memory rather than the three required by the LXI instruction, which could also have been used for this purpose. Also noteworthy is the use of the increment and decrement instructions.

As an indication that these kinds of optimizations may not be as easy to apply as it might at first appear, consider Figure 20. This figure shows the PDP-11 machine code [15] generated for the two functionally equivalent sets of C language statements discussed in Section III.B. It can be seen that, while the compiler has used increment and decrement instructions in both cases, the code in Figure 20(b) is less efficient than the other, even though it has been passed through the optimizer associated with the C compiler. (In fairness, it should be noted that the optimizer is claimed to be only experimental.) This points up the fact that machine-dependent optimizations tend to be applied in

```
                    i = ++j - k;
                    i = j-- - k;

                    -----------

                    inc     j
                    mov     j,r0
                    sub     k,r0
                    mov     r0,i
                    mov     j,r0
                    sub     k,r0
                    dec     j
                    mov     r0,i

                        (a)


            i = (j = j + 1) - k;
            i = j - k; j = j - 1;

            ---------------------

                    mov     j,r0
                    inc     r0
                    mov     r0,j
                    sub     k,r0
                    mov     r0,i
                    mov     j,r0
                    sub·    k,r0
                    mov     r0,i
                    mov     j,r0
                    dec     r0
                    mov     r0,j

                        (b)
```

Figure 20. PDP-11 assembly code for two
equivalent sets of C language statements
(a) using increment/decrement feature,
(b) using addition and subtraction

an ad hoc manner, that is, by testing a series of conditions which would indicate special cases in the code being generated.

There is little, if any, mathematical rigor associated with these methods, and they thus are very similar to the kinds of optimizations which an assembly language programmer would make. This is the major reason for the crossover in relative efficiency between assembly language and high-level language programs as program size increases (see Section II.A). For the high-level language the special cases must be foreseen by the compiler writer. Since he probably will overlook some, the compiler will generate some code which is obviously inefficient, as in Figure 20(b). Nevertheless, those optimizations which can be applied to cases foreseen by the compiler designer will be applied consistently by the compiler every time the appropriate conditions are satisfied. The assembly language programmer, on the other hand, will easily spot the kinds of inefficiencies shown in the example (and in the example of Figures 15-17), but he may not be consistent in applying optimizations and may not recognize others because of the complexity of the program. The inefficiencies contributed by these two factors tend to build up rapidly as the assembly language program increases in size.

2. Architecture-Dependent

Architecture-dependent optimizations are global in nature and depend on the architecture of the object machine

113

but not the instruction set. Examples of architectural features which are considered are the number of registers, the number of processing elements, and the degree of pipelining. As can be seen, these types of optimizations generally involve resource allocation and thus would be very important in compiling for microprogrammable and modular systems. The reason these are considered to be global optimizations is that the resource requirements of diverse segments of a program must be considered when making the allocations.

The important register-allocation problem fits into this category, since its solution depends on the numbers and types of registers available in the architcture. The particular machine instructions are not important in this case. It should be recalled that poor register allocation was one of the major causes of inefficiency in the code of Figure 17. Algorithmic solutions have been found for the register-allocation problem for simple straight-line (non-looping) programs [52], but a general solution is either not possible or not practical. The former is usually the case in programs which contain conditional branches, since the flow of execution of the program is almost always unknown at compile time, and this information is needed for an optimal solution. The latter is usually true for long programs, even if they contain no loops, since an optimal solution would require an analysis of the entire program and an enumeration of all possible combinations of register

assignments. Freiburghouse [20] has recently presented a method for solving the register-allocation problem which takes advantage of information which can be accumulated during the normal course of compilation and which appears to give results closer to the optimum than other proposed solutions.

As discussed in Section VI.C, parallelism is an important feature in firmware systems, and the generation of code to take maximum advantage of this parallelism is another architecture-dependent optimization problem. An important use of parallelism in improving execution of a program lies in the area of reducing the time required for iterative segments of code. For example, the PL/M code of Figure 21 could be translated into more time-efficient code if several arithmetic units were available than if only one were available.

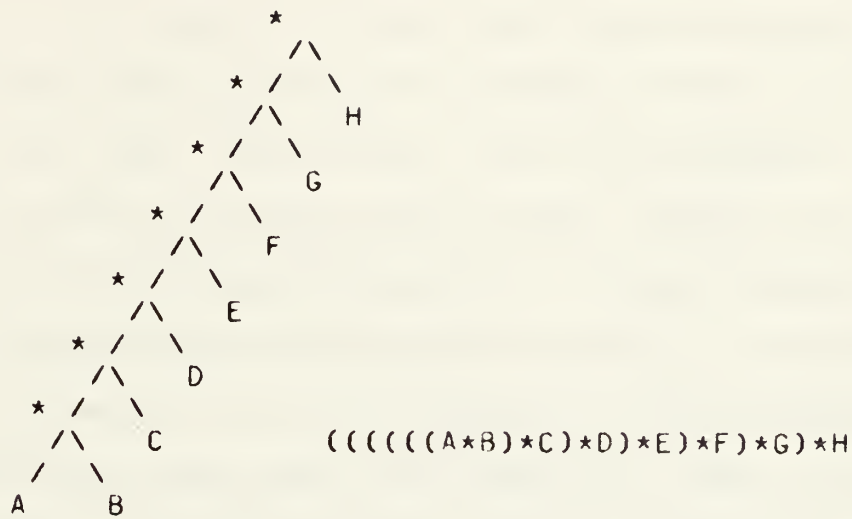
```
DO I = 1 TO 20;
A = (B(I) + B(I+1)) * 2;
C(I) = C(I) + A;
D(I) = D(I) - A;
END;
```

Figure 21. Iterative code for which
parallel processing would be useful

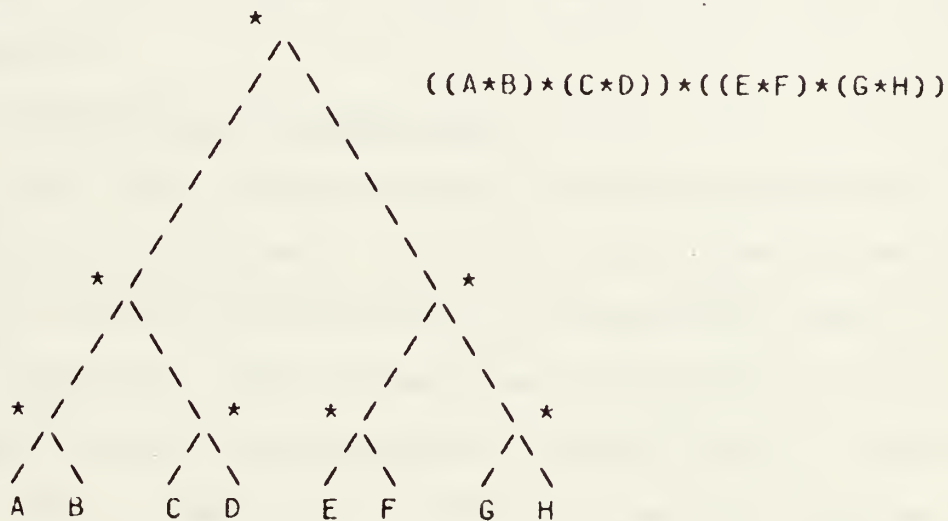---

As is usually the case in optimization problems, there are tradeoffs which must be made when dealing with parallelism. The two methods shown in Figure 22 for

115

```
                              *
                             /\  \
                        * /    \
                         /\     H
                    * /    \
                     /\     G
                * /    \
                 /\     F
            * /    \
             /\     E
        * /    \
         /\     D
    * /    \
     /\     C        ((((((( A*B ) *C ) *D ) *E ) *F ) *G ) *H
   /  \
  A    B
```

(a)

```
                    *
                   /\
                 /    \
               /        \
             /            \
           /                \
         /                    \
       * /                    \ *
        /\                    /\
      /    \                /    \
    /        \            /        \
  * /        \ *        * /        \ *
   /\          /\          /\          /\
  /  \        /  \        /  \        /  \
 A    B      C    D      E    F      G    H
```

((A*B) * (C*D)) * ((E*F) * (G*H))

(b)

Figure 22.   Tree structure for serial and
parallel computation of an expression. (a) Tree yielding
minimum number of registers, (b) Tree yielding
maximum inherent parallelism [52, p.2]
```

calculating an expression can be used to illustrate this point. If code were being generated for a machine with only one register the scheme in Figure 22(a) would be better than that of Figure 22(b), while the reverse would be true for a machine with four multipliers and four registers. For a machine with fewer than four multipliers, though, it is not obvious which method would be better. In such situations an analysis must be made of the various types of instructions involved. One way to do this would be to assign weights to the instructions based upon their execution times (e.g., a multiplication instruction would have a greater weight than an addition instruction) and then generate the code which achieved the minimum total weight for the desired computation.

3. Architecture-Independent

The final and most general category consists of the architecture-independent optimizations. Since these do not depend on the architecture or the instruction set of the object machine, they are obviously applicable to compiling for user-definable architectures. These kinds of optimizations can be applied to the intermediate language code without considering the hardware features available. As in the case of architecture-dependent optimization, these optimizations are global in nature. The most commonly applied techniques in this class are common subexpression elimination, dead variable elimination, code motion, and constant propagation. Since these techniques are widely discussed in

117

the literature (see, e.g., [4] for a good survey and [13]
for an application) only a brief description is presented
here.

Common subexpression elimination is the most widely
employed technique [52]. Basicly it is concerned with
avoiding redundant computations, such as for the second
occurrence of "B*C" in Figure 23(a). Dead variables are
those which, beyond a given statement, never again appear on
the right-hand side of an assignment or are never again
referenced. In the first case the variable need not be kept
in a high-speed register, and in the second case it need not
any longer be assigned any memory at all. Code motion
refers to the movement of sections of code so as to reduce
the execution time of a program. For example, the section
of code shown in Figure 23(b) would be significantly im-
proved if the assignment to "D" were moved outside of the
loop. Constant propagation is really a special case of code
motion, since calculations involving only known constants
are moved from the execution phase of a program to the com-
pilation phase. The computations of "C" and "D" in Figure
23(c) provide examples of propagated constants.

Architecture-independent optimization techniques
rely heavily on theoretical work and are amenable to the
application of sophisticated algorithms. They usually in-
volve a global flow analysis of the intermediate form of the
program and may rely on graph theory or matrix analysis.
Unfortunately most of these techniques are very complicated
and require large amounts of memory and time.

```
A = B * C + D;
Q = D + R;
X = P + B * C;
```

(a)

```
DO I = 1 TO 1000;
A(I) = B(I) * C(I);
D = X * Y / Z + 50;
B(I) = C(I) * D;
END;
```

(b)

```
A = 3;
B = C * D;
C = A + 5;
D = A * C + 4;
```

(c)

Figure 23.   Architecture-independent
optimization candidates. (a) Common subexpression,
(b) Code motion, (c) Constant propagation

Since a useful program usually contains many branches and loops which make it impossible to know at compilation time how often (if ever) many sections of code will be executed, frequency analysis is sometimes employed in the optimization process. By assigning a relative frequency or weight to each block of code in a program, the programmer allows the optimizer to perform a Monte Carlo simulation to determine the "optimum" code sequence [52]. There have even been proposals [24] to employ an adaptive optimization process to perform the optimizations at run time. In such a scheme a large portion of the effort would be devoted to optimizing sections of code which are heavily used, since they account for most of the execution time. Such a scheme probably would not be practical for most real-time systems unless the adaptive optimization were done during the development process and the resulting optimizations were applied to the final system in a non-adaptive mode.

C. APPLICATION

From the discussion in Sections VII.A and VII.B it can be seen that compiler optimization is a complex problem. A good optimizing compiler, in effect, attempts to match wits with a good assembly language programmer. In order to do this effectively the compiler must have a great deal of "artificial intelligence" built into it, and this is something which, unfortunately, is difficult to do. "Optimizations originating in the academic and scientific community tend to be global, while, until recently, manufacturers have

120

concentrated on local and machine-dependent techniques."
[52, p.1] More efficient algorithms must be developed in
order to allow the academic solutions to become more useful
in practical compilers. More general and powerful tech-
niques for handling local and machine-dependent optimiza-
tions must also be found. For the types of systems under
consideration here, many of the techniques discussed above
are already practical, since compilation costs are only a
small part of total development cost.

A great deal of care must be exercised in the applica-
tion of optimization techniques in code generation. Many of
the techniques involve reordering of arithmetic operations,
and this can lead to unexpected and often undesired results
(e.g., from a numerical analysis point of view). Thus it
appears that a great deal of work remains to be done in this
area. It is evident, though, that as better techniques are
developed and the cost of current techniques (in memory and
time) are brought lower, high-level languages will continue
to become more attractive.

Until some breakthrough comes in the artificial intelli-
gence area the most practical techniques will probably
require the programmer to provide some input to the optimi-
zation process. He might specify that speed is most impor-
tant for certain sections of code and that the amount of
memory utilized should be minimized for other sections. He
might also specify the probabilities of certain branches in
the program (as has been done in some compilers since the

121

1950's [52]). The computer then will perform the "dirty work" much more effectively than a human writing in assembly language.

VIII.   THE CONFIGURATION-INDEPENDENT COMPILER

The previous seven chapters have discussed features available in current compilers and features which appear feasible for future compilers. In this chapter an attempt is made to tie together some of these ideas and discuss the possible functioning and structure of a compiler for which a target machine and language are not necessarily specified prior to compilation. The level of interest in developing such a compiler is indicated by the increasing amount of work being done on machine-independent high-level microprogramming and system programming languages [18,38,40,47,57].

Ramamoorthy and Tsuchiya [47] have demonstrated a language which appears to have many of the desired features and which can produce control code for a complex microprogrammable machine. Their SIMPL (Single Identity Microprogramming Language) is intended to be machine-independent; however, it does not appear that they have yet addressed the problem of specifying the machine organization to the compiler in a flexible manner.

Wilcox [61] has looked at the latter problem but has based his work on the concept of a machine-independent assembler. This assembler is to be used for generating control code for digital systems built with QED functional modules [56]. The nature of this problem is very similar to that considered by Ramamoorthy and Tsuchiya, and there does

not seem to be any practical reason for not extending Wilcox's concept to a machine-independent compiler.

A. THE IDEAL COMPILER

The truly ideal compiler would be one which would accept an algorithm from the programmer in a universal programming language, select the most appropriate hardware for the job, and produce the code for controlling the hardware. Obviously the compiler would require more input than just a statement of the algorithm. It would need to have information on what hardware was available and the operational constraints to be placed on the resulting system.

A compiler which could function as described above is a goal for which compiler designers can strive, but it is one which will probably require many more years to achieve. The reason for this is the reason that computers have not taken over all other engineering disciplines--there are too many subtle tradeoffs to be made in designing a system. The relationships between many of the variables involved cannot be quantified, and a great deal of experience and intuition is required to produce a good design. A large part of any design effort is concerned with optimization of some sort, and, as discussed in Chapter VII, this involves the area which the computer scientist labels artificial intelligence.

Short of the ideal, the programmer (system designer) will have to specify a few possible hardware configurations along with the optimization functions and constraints. The compiler will then make some simple tradeoffs among the

various configurations, choose the "optimal" one, and produce the optimal code. In a given design, for example, the compiler might decide that a system with three multiplier modules would be better than one with two or four multiplier modules.

It will probably be several years before even this reduced capability compiler can be implemented. Based upon what appears feasible within the next few years, the "ideal" compiler would be even more restricted. As indicated in Section I.A this compiler would have several inputs. In addition to the algorithm, the programmer would specify the hardware configuration, the format of the control code, and some simple optimization information. A conceptual block diagram of such a compiler is shown in Figure 24. In actual practice it may be difficult to divide the compiler into a set of neat boxes with definite flow of action, a fact which is suggested by the dashed line in Figure 24. In other words, there will probably be a strong interaction among the various sections of such a compiler.

It will probably be especially difficult to distinguish the architecture-dependent optimization phase from pass 2. These two phases relate fairly closely to the final two steps in a SIMPL compilation--the concurrency and timing analysis step and the microoperation timing optimization step. (The first two steps are syntactic analysis and semantic analysis, which parse the source program and break it into a series of subblocks.). The concurrency and timing

```
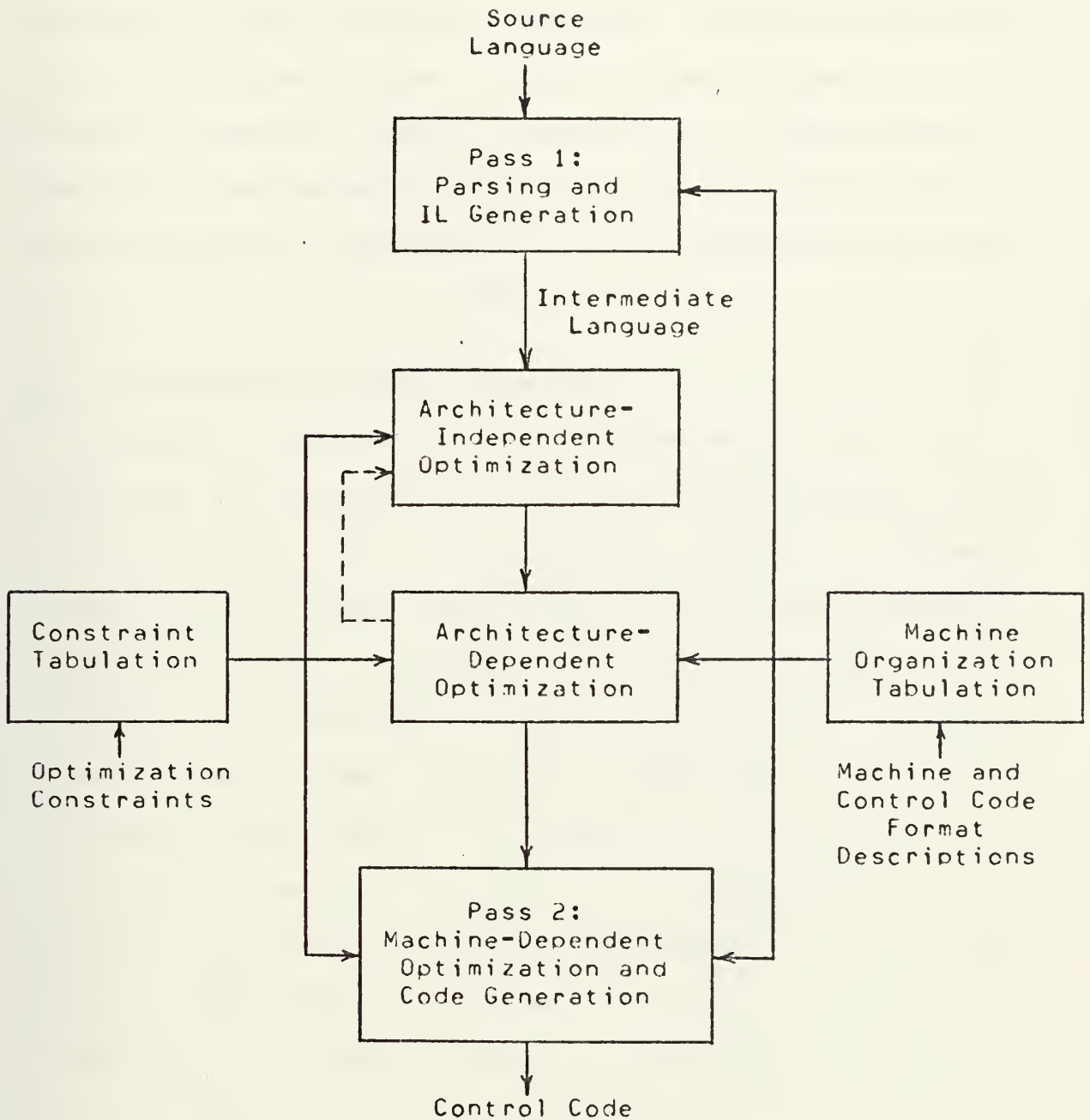   ┌─────────────────┐
   │  Source         │
   │  Language       │
   └────────┬────────┘
            │
            ▼
   ┌─────────────────┐
   │  Pass 1:        │◄──────────┐
   │  Parsing and    │           │
   │  IL Generation  │           │
   └────────┬────────┘           │
            │                    │
            ▼ Intermediate       │
              Language           │
   ┌─────────────────┐           │
   │  Architecture-  │           │
   │  Independent    │           │
   │  Optimization   │           │
   └────────┬────────┘           │
            │                    │
            ▼                    │
   ┌─────────────────┐  ┌────────┴────────┐
   │  Architecture-  │◄─│  Machine        │
   │  Dependent      │  │  Organization   │
   │  Optimization   │  │  Tabulation     │
   └────────┬────────┘  └─────────────────┘
```

Figure 24.   Conceptual diagram of a compiler
          for user-definable architectures

analysis step "... examines symbolic code in each subblock to detect concurrently executable microoperations and to determine their feasible execution timing." [47, p.796] Next, the microoperation timing optimization step "... introduces complete machine dependence.... The hardware organization and operating characteristics are defined by the microinstruction definition that is represented internally in the compiler." [47, p.797]

B. INTRODUCING MACHINE DEPENDENCE

Probably the most difficult problem which will be encountered in designing a compiler for user-definable architectures will be that of introducing machine dependence. Compilers for fixed architectures have machine-dependent information scattered through all of their phases. The configuration-independent compiler, on the other hand, must have machine-dependent information localized to as few areas as possible and must be structured in such a way as to make it as easy as possible to change this information. It is because of the fact that machine dependence has to be introduced at some stage in any practical compiler that the term "compiling for user-definable architectures" has been used in this thesis. The technically inaccurate term "machine-independent compiler" is often encountered and has the same meaning.

As indicated in Figure 24, information on optimization, machine organization, and instruction formats will be tabulated by the compiler. After suitable processing, the

127

information will be loaded into tables in much the same way that the information from the algorithm is loaded into the symbol table. The architecture-dependent optimization phase and pass 2 will thus be "table-driven"; i.e., they will extract information from the various tables and use this information to make optimization and code generation decisions. In the sense that they use information from the symbol table to generate control code, the second passes of the two current Intel PL/M compilers for the 8008 and the 8080 microprocessors can be considered to be partially table-driven.

Tirrell [57] has reported work involving the use of a table-driven compiler for microprogramming. In his compiler, tables containing machine-dependent information could be loaded prior to compilation or could be generated during compilation. One table was used for indicating the status of the various hardware registers and indicators, while a second table was used to store the basic microinstruction patterns. Other tables were used as aids in optimizing the generated code. Most of the optimizations involved the arrangement of elementary operations into efficient microinstruction words (i.e., words which take maximum advantage of parallelism).

Another concept which deserves attention in the design of a compiler for user-definable architectures is that of decision-logic tables [5,44]. Initially conceived to replace flow charts in business programming applications,

128

decision tables have developed an extensive body of theory to enable their efficient use. A decision table consists of a group of alternatives for a given situation and a set of actions to be taken for each alternative. In essence, this technique results in a tabular program rather than a table-driven program.

In his discussion of register-transfer languages, Barbacci concluded with some very pertinent remarks.

The proliferation of machines introduces a problem in the production of software. Standard languages (Fortran, Cobol, etc.) have alleviated the problem with respect to the user side but there still remains the variability on the machine side. Compiler writing is an expensive task and automatic programming systems (compiler-compilers) have not taken into account this variability on the target machine. If we expect to solve the problem we need compiler-compilers that accept as inputs both the language description (syntax and semantics) and the target machine description. None of the existing hardware design languages is useful in this problem and the issue is not just the production of code, but of "good" code .... [6, p.148]

IX. CONCLUSIONS AND RECOMMENDATIONS

As digital large scale integrated circuits and function-
al modules continue to have a greater impact on electronic
system design, the need for improved software design and
application will become ever more critical in producing
reliable, cost-effective systems. Most of the concepts dis-
cussed in this thesis have been in existence for a number of
years, but current hardware development trends demand that
greater emphasis be placed on translating these concepts
into realities.

One of the key milestones in the effort to provide
better tools for the design of systems using the new com-
ponents will be the development of suitable high-level pro-
gramming languages for describing the algorithm. There are
well over 100 high-level languages available today, each
designed to help solve a particular problem. "... [O]ne may
question the need or desirability of all these languages.
On the other hand, for the convenience of the user, he
should be allowed to choose a language that he is comfort-
able with and which best suits his application." [48, p.3]
The PL/M language, developed by Intel Corporation, has been
successfully used by firmware designers and may be able to
be used as a base for new, more comprehensive languages.
Even if completely new languages are developed, they will
probably bear a strong resemblance to PL/M.

Major effort will have to be directed toward the development of compiler facilities which allow user specification of the hardware aspects of his design. This will require the development of a good hardware description language, which may or may not be a subset of the language discussed above for describing the algorithm. In any event, the compiler will have the capability of manipulating this hardware information in such a way as to facilitate the generation of control code.

In order for this compiler to be accepted by system designers, it will have to generate "good" control code, with the specification of goodness being provided by the user. Thus there is a need for the continued development of practical compiler optimization techniques. In all of the work to be done, the optimization problems will probably be the most difficult to solve and the most crucial for the success of the overall task.

The work discussed in Chapter IV has been sufficient to indicate the feasibility of developing a high-level language for user-definable architectures; however, there are many questions left to be answered and several important steps which need to be taken. The development of a formal technique for describing the semantics of a programming language should have a high priority in this regard. Despite all of the thoretical work which has been done to improve the syntax analysis and parsing processes in compilers, very little has been done to formalize the semantic analysis and code

generation processes [30,60]. The code generation process for the PL/M compiler is just another translation (from intermediate language to machine language) and might be able to be performed in a manner similar to the parsing of the source language and generation of intermediate language code. The use of a push-down automaton for this process should be investigated.

Error recovery during source language parsing is another area which deserves additional attention. It is desirable to provide the programmer with as much information as possible, and the method discussed in Section IV.B.4 is relatively simple. Attention in this area should also be devoted toward more efficient storage of error messages in order to help minimize the size of the compiler. One technique for doing this would involve the design of messages which can be partitioned into a relatively small number of common phrases. Detailed messages could then be constructed from these phrases.

The next step in continuing the work described in Chapter IV should be the design and implementation of a second pass for the PL/M compiler. Several specific recommendations can be made here for future work in this area. First, a routine will have to be written to transfer the symbol table to a disk file. This file, along with the intermediate language file and the initial value file, would then be used as input for the second pass. The key to successful development of a "machine-independent" second pass

will be the availability of a suitable hardware description language and compiler. When they become availble, they should be tested by using them to write a description of the Intel 8008 or 8080 microprocessor. In order to produce the control code, routines will then have to be written to store the necessary information from this description in tables and to manipulate these tables according to the information received from pass 1. Until a suitable hardware description language is available, a more conventional pass 2 could be written, in the C language, for the PL/M compiler. This would provide a vehicle for testing various optimization techniques. Finally, optimization inputs must be defined, and the methods for utilizing them must be developed.

APPENDIX A
PL/M INTERMEDIATE LANGUAGE CODES

Prefixes

```
ADR ..... Load Address of Symbol
LIN ..... Line Number Marker
LIT ..... Load Literal Value
OPR ..... Stack Operator
VAL ..... Symbol: Load Value
          Procedure: Load Address
```

Operators

```
ADC ..... Add with Carry
ADD ..... Add
AND ..... Logical And
ARG ..... Procedure Argument
AX1 ..... Auxiliary 1
AX2 ..... Auxiliary 2
AX3 ..... Auxiliary 3
BIF ..... Built-In Function
CSE ..... Case Index Operation
CVA ..... Convert to Address (Double Byte)
DAT ..... Data Start/Finish
DEL ..... Delete
DIS ..... Disable Interrupts
DIV ..... Divide
DRT ..... Default Return (End of Procedure)
ENA ..... Enable Interrupts
ENB ..... Enter Block
END ..... End of Do Group
ENP ..... Enter Procedure
EQL ..... Test for Equal
GEQ ..... Test for Greater Than or Equal
GTR ..... Test for Greater Than
HAL ..... Halt
HIV ..... Extract High Order Byte
INC ..... Increment
INX ..... Subscript Index
IOR ..... Logical Inclusive Or
LEQ ..... Test for Less Than or Equal
LOD ..... Load
LOV ..... Extract Low Order Byte
LSS ..... Test for Less Than
MUL ..... Multiply
NEG ..... Negative
NEQ ..... Test for Not Equal
```

```
NOP  .....  No Operation
NOT  .....  Logical Negate
ORG  .....  Origin
PRO  .....  Procedure Call
REM  .....  Remainder
RET  .....  Return
RTL  .....  Rotate Left
RTR  .....  Rotate Right
SBC  .....  Subtract with Carry
SFL  .....  Shift Left
SFR  .....  Shift Right
STD  .....  Store Destructive
STO  .....  Store
SUB  .....  Subtract
TRA  .....  Unconditional Transfer
TRC  .....  Conditional Transfer
XCH  .....  Exchange
XOR  .....  Logical Exclusive Or
```

```
%term identifier number string

%{        /* declarations used by actions and programs */

int ii,jj; char *kk,tt;

#include "m.def"
#include "m.decl"

%}

%%        /* beginning of grammar rules section */

program: statementlist  /* 1 */
;
statementlist: statement    /* 2 */
|      statementlist statement   /* 3 */
;
statement:              basicstatement  /* 4 */
        =    {npush = 0; }
|                        ifstatement   /* 5 */
        =    {npush = 0; }
;
basicstatement:    assignment ';'    /* 6 */
        =    {while ($1--)
                {if (fixv[sp] > 0) emit(OPR,XCH);
                else {setsy(symloc[sp]);
                    emit(ADR,getsyno());}
                pop(1);
                if ($1 > 0) emit(OPR,STO);
                else emit(OPR,STD); } }
|                     group ';'       /* 7 */
|                     proceduredefinition ';'   /* 8 */
|                     returnstatement ';'    /* 9 */
|                     callstatement ';'       /* 10 */
|                     gotostatement ';'       /* 11 */
|                     declarationstatement';'    /* 12 */
|                     'halt'  ';'    /* 13 */
        =    {emit(OPR,HAL); }
|        'enable' ';'      /* 14 */
        =    {emit(OPR,ENA); }
|        'disable' ';'     /* 15 */
        =    {emit(OPR,DIS); }
```

```
|                      ';'    /* 16 */
|    labeldefinition  basicstatement /* 17 */
|                error ';'          /* ERROR */ /* 18 */
            =      {errfix();}
;
ifstatement:            ifclause statement    /* 19 */
        =    {emit(DEF,spop()); }
|                    ifclause truepart statement    /* 20 */
        =    {emit(DEF,spop()); }
|    labeldefinition  ifstatement      /* 21 */
;
ifclause:    'if' expression 'then'  /* 22 */
        =    {emit(VAL,spush(nsym++));
            emit(OPR,TRC); }
;
truepart:               basicstatement 'else'     /* 23 */
        =    {ii = spop();
            emit(VAL,spush(nsym++));
            emit(OPR,TRA);
            emit(DEF,ii); }
;
group:               grouphead ending     /* 24 */
        =    {exitblk();
            if ($2 >= 0) {flag("identifier invalid here");
                pop(1);}
            switch($1 & 03)
                {case 0:     /* simple group */
                    emit(OPR,END); break;
                case 1: if ($1 & 04)    /* stepdef w/BY */
                        {emit(VAL,spop());
                        emit(OPR,TRA); emit(DEF,spop());}
                    else    /* stepdef w/o BY */
                        {emit(VAL,ii = symfind(sp));
                        emit(OPR,INC);
                        emit(ADR,ii); emit(OPR,STD);
                        ii = spop(); emit(VAL,spop());
                        emit(OPR,TRA); emit(DEF,ii);}
                    pop(1); break;
                case 2:     /* while group */
                    ii = spop(); emit(VAL,spop());
                    emit(OPR,TRA); emit(DEF,ii); break;
                case 3:      /* case group */
                    ii = spop(); spop(); jj = $1 >> 2;
                    kk = csp + (jj << 1);
                    emit(DEF,maketwo(*kk,*(kk+1)));
                    emit(OPR,CSE);
                    while (jj--)
                        {kk =- 2;
                        emit(VAL,maketwo(*kk,*(kk+1)));
                        emit(OPR,AX2);}
                    emit(DEF,ii);
                    for (jj = ($1 >> 2) + 1;jj--;) spop();
                    break; } }.
;
```

```
grouphead:              'do' ';'         /* 25 */
        =       {enterblk(); emit(OPR,ENB); $$ = 0; }
|                       'do' stepdefinition ';'    /* 26 */
        =       {enterblk(); $$ = ++$2; }
|   'do' whileclause ';'     /* 27 */
        =       {enterblk(); $$ = 2; }
|   'do' caseselector ';'    /* 28 */
        =       {enterblk(); $$ = 3;
                emit(VAL,spush(nsym++)); emit(OPR,AX1);
                emit(DEF,spush(nsym++)); spush(nsym++); }
|                       grouphead statement    /* 29 */
        =       {if ((($$ = $1) & 03) == 3)
                {emit(VAL,ii = spop()); emit(OPR,TRA);
                emit(DEF,spush(nsym++));
                spush(ii); $$ =+ 4;} }
;
stepdefinition: variable replace expression iterationcontrol
                                                /* 30 */
        =       {$$ = $4; }
;
iterationcontrol: to expression /* 31 */
        =       {$$ = 0; emit(OPR,LEQ);
                emit(VAL,spush(nsym++)); emit(OPR,TRC); }
|   to expression by expression  /* 32 */
        =       {emit(VAL,ii = symfind(sp)); emit(OPR,ADD);
                emit(ADR,ii); emit(OPR,STD); $$ = 4;
                emit(VAL,spop()); emit(OPR,TRA);
                emit(DEF,spop()); }
;
whileclause: while expression    /* 33 */
        =       {emit(VAL,spush(nsym++));
                emit(OPR,TRC); }
;
caseselector:  'case' expression      /* 34 */
;
proceduredefinition: procedurehead statementlist ending
                                                /* 35 */
        =       {if ($3 < 0) flag("identifier required");
                else {setsy($1); ii = getsyno();
                    if (ii != symfind($3))
                        flag("incorrect identifier");
                    pop(1);}
                exitblk(); emit(OPR,END);
                emit(OPR,DRT); emit(DEF,spop()); }
;
procedurehead:  procedurename ';'     /* 36 */
        =       {procode($$ = $1); }
|                   procedurename type ';'     /* 37 */
        =       {setsy($$ = $1); setprec($2); procode($1); }
|   procedurename parameterlist';'    /* 38 */
        =       {setsy($$ = $1); setlen($2); procode($1); }
|   procedurename parameterlist type ';'     /* 39 */
        =       {setsy($$ = $1); setlen($2);
                setprec($3); procode($1); }
```

138

```
;       procedurename 'interrupt' number ';'    /* 40 */
        =      {procode($$ = $1); }
;
procedurename:   identifier ':' 'procedure'     /* 41 */
        =      {if (symloc[$1] >= curlev)
                   flag("illegal procedure name");
               fixv[$1] = 0; $$ = sytop;
               enter($1,prot,0,0); compress($$,1);
               pop(1); emit(OPR,ENP);
               enterblk();}
;
parameterlist:   parameterhead identifier ')'     /* 42 */
        =      {if (acnt >= 63)
                   {flag("too many parameters"); acnt = 62;}
               setsy($1);
               fixv[$2] = 0; $$ = ++acnt + (getlast() << 6);
               enter($2,undef,0,1); compress($1,acnt);
               pop(1);}
;
parameterhead:   '('    /* 43 */
        =      {$$ = sytop; acnt = 0; }
;    parameterhead  identifier ','     /* 44 */
        =      {$$ = $1; acnt++; fixv[$2] = 0;
               enter($2,undef,0,1);
               pop(1);}
;
ending:                'end'     /* 45 */
        =      {$$ = -1; }
;                      'end' identifier   /* 46 */
        =      {$$ = $2; }
;    labeldefinition ending   /* 47 */
        =      {$$ = $2; }
;
labeldefinition:   identifier ':'      /* 48 */
        =      {labflag++;
               if ((ii = symloc[$1]) >= curlev)
                   {setsy(ii);
                   if (getlen())
                       {ii = getsize() + finfo + 1;
                       *(symbol + ii) =& 03;
                       emit(DEF,getsyno());
                       }
                   else flag("label redeclared");
                   }
               else
                   {fixv[$1] = 0;
                   $$ = sytop;
                   enter($1,labt,0,0); compress($$,1);
                   emit(DEF,nsym-1);
                   }    }
;      number ':'    /* 49 */
        =      {emit(LIT,$1); emit(OPR,ORG); }
;
returnstatement:   'return'      /* 50 */
```

139

```
             =      {emit(LIT,0); emit(OPR,RET); }
         ;                  'return' expression    /* 51 */
             =      {emit(OPR,RET); }
         ;
callstatement:        'call' variable   /* 52 */
         =      {setsy(symloc[$2]); emit(VAL,getsyno());
                if ((ii = gettype()) == prot) emit(OPR,PRO);
                else {if (ii == cprot) emit(OPR,BIF);
                    else flag("variable not a procedure name");}
                pop(1);   }
         ;
gotostatement:      goto identifier    /* 53 */
             =      {emit(VAL,symfind($2)); emit(OPR,TRA); pop(1);  }
         ;   goto number /* 54 */
             =      {emit(LIT,$2); emit(OPR,TRA);  }
         ;
goto:                    'go' 'to'      /* 55 */
         ;                  'goto'      /* 56 */
         ;
declarationstatement: 'declare' declarationelement /* 57 */
         ;      declarationstatement ',' declarationelement   /* 58 */
         ;
declarationelement:   typedeclaration       /* 59 */
         =      {setsy(curlev - *curlev);
                if (gettype() == prot)
                    {ii = getlen();
                    while (ii--)
                        {setsy(symbol + finfo + getsize() + 2);
                        if (gettype() == vart && getorec() == 0)
                            setprec($1);
                        }
                    }   }
         ;   identifier 'literally' string   /* 60 */
         =    {
             /* enter a macro definition */
             symbol = sytop;
             setname($1);     /* fills size,name,hcoll */
             fixhcoll();
             settype(mact);
             /* set the macro definition size */
             setsym((ii = getsize() + finfo),
                 getvarc(jj = var[$3]));
             setchar(++ii,jj);    /* fills the macro definition */
             /* note that last field filled is at end of entry */
             syfin();
             pop(2);         }
         ;    identifier datalist    /* 61 */
         =      {if (symcheck($1))
                    {fixv[$1] = 0; ii = sytop; jj = ($2 > 63);
                    enter($1, jj ? lvect : svect,1,$2);}
                if (!jj) compress(ii,1);
                else {setsy(ii); fixhcoll();}
                pop(1); emit(OPR,DAT); emit(DEF,spop());  }
         ;
```

140

```
datalist: datahead constant ')'        /* 62 */
        =    {$$ = $1 + concode($2,datcon);   }
;
datahead:   'data' '('     /* 63 */
        . =    {$$ = 0; emit(VAL,spush(nsym++));
               emit(OPR,TRA); emit(OPR,DAT);
               emit(DEF,nsym);  }
|    datahead constant ','     /* 64 */
        =    {$$ = $1 + concode($2,datcon);   }
;
typedeclaration: identifierspecification type    /* 65 */
        =    {$$ = $2; ii = $1;
              if ($2 != 1) change(vart,$2,1,acnt);
              compress($1,acnt);}
|    boundhead number ')' type     /* 66 */
        =    {if (!$4) flag("illegal declaration");
              $$ = $4; ii = $1;
              if ($2 > 63) change(lvect,$4,$2,acnt);
              else {change(svect,$4,$2,acnt);
                  compress($1,acnt);}}
|    typedeclaration initiallist /* 67 */
        =    {$$ = $1;}
;
type:                  'byte'   /* 68 */
        =    {$$ = tt = 1;}
|    'address'  /* 69 */
        =    {$$ = tt = 2;}
|    'label'       /* 70 */
        =    {$$ = tt = 0;}
;
boundhead: identifierspecification '('   /* 71 */
        =    {$$ = $1;}
;
identifierspecification: variablename    /* 72 */
        =    {$$ = $1; if (jj) acnt = 1; else acnt = 0; }
|    identifierlist variablename ')'     /* 73 */
        =    {if (acnt++) $$ = $1; else $$ = $2;}
;
identifierlist: '('      /* 74 */
        =    {acnt = 0;}
|    identifierlist variablename ','      /* 75 */
        =    {if (acnt++) $$ = $1; else $$ = $2;}
;
variablename: identifier     /* 76 */
        =    {if (symcheck($1))
                  {fixv[$1] = 0;
                  $$ = sytop;
                  enter($1,vart,1,1);}
              pop(1);}
|    basedvariable identifier    /* 77 */
        =    {if (fixv[$2] != foundv)
                  flag("base not defined");
              else {ii = getsyno();
                  setsy($1); setbsyno(ii); }
```

141

```
                $$ = $1;
                pop(1);}
    ;
    basedvariable: identifier 'based'    /* 78 */
          . =    {if (symcheck($1))
                   {fixv[$1] = basev;
                    $$ = sytop;
                    enter($1,vart,1,1);}
                pop(1);}
    ;
    initiallist:        initialhead constant ')'       /* 79 */
            =    {$$ = $1 + concode($2,tt);
                if (tt)
                   {putw($$,&buf3); setsy(ii);
                        putw(getsyno(),&buf3);} }
    ;
    initialhead:        'initial' '('       /* 80 */
            =    {$$ = 0; if (!tt)
                    flag("initial not allowed here"); }
    :                initialhead constant ','  /* 81 */
            =    {$$ = $1 + concode($2,tt);   }
    ;
    assignment:  variable replace expression      /* 82 */
            =    {$$ = 1; }
    :                        leftpart assignment    /* 83 */
            =    {$$ = ++$2; }
    ;
    replace:  '='    /* 84 */
    ;
    leftpart:            variable ','       /* 85 */
    ;
    expression: logicalexpression    /* 86 */
    :     variable ':' '=' logicalexpression      /* 87 */
            =    {if (fixv[$1]) emit(OPR,XCH);
                else emit(ADR,symfind($1));
                emit(OPR,STO); pop(1); }
    ;
    logicalexpression: logicalfactor      /* 88 */
    :    logicalexpression 'or' logicalfactor    /* 89 */
            =    {emit(OPR,IOR); }
    :    logicalexpression 'xor' logicalfactor    /* 90 */
            =    {emit(OPR,XOR); }
    ;
    logicalfactor:      logicalsecondary /* 91 */
    :       logicalfactor 'and' logicalsecondary      /* 92 */
            =    {emit(OPR,AND); }
    ;
    logicalsecondary:  logicalprimary    /* 93 */
    :    'not' logicalprimary    /* 94 */
            =    {emit(OPR,NOT); }
    ;
    logicalprimary: arithmeticexpression    /* 95 */
    : arithmeticexpression relation arithmeticexpression/* 96 */
            =    {emit(OPR, $2); }
```

```
;
relation:              '='         /* 97 */
        =      {$$ = EQL; }
|    '<'  /* 98 */
        =      {$$ = LSS; }
|    '>'  /* 99 */
        =      {$$ = GTR; }
|    '<''>'   /* 100 */
        =      {$$ = NEQ; }
|                       '<''='      /* 101 */
        =      {$$ = LEQ; }
|                       '>' '='     /* 102 */
        =      {$$ = GEQ; }
;
arithmeticexpression: term   /* 103 */
|                 arithmeticexpression '+' term /* 104 */
        =     {emit(OPR,ADD); }
|                 arithmeticexpression '-' term /* 105 */
        =     {emit(OPR,SUB); }
|     arithmeticexpression 'plus' term    /* 106 */
        =     {emit(OPR,ADC); }
|     arithmeticexpression 'minus' term  /* 107 */
        =     {emit(OPR,SBC); }
|     '-' term   /* 108 */
        =     {emit(OPR,NEG); }
;
term: primary    /* 109 */
|    term '*' primary     /* 110 */
        =     {emit(OPR,MUL); }
|    term '/' primary     /* 111 */
        =     {emit(OPR,DIV); }
|    term 'mod' primary   /* 112 */
        =     {emit(OPR,REM); }
;
primary: constant    /* 113 */
        =     {if (conlast == stringc)
                  {ii = var[sp] + 1;
                   switch ($1)
                        {case 1: emit(LIT,getvarc(ii)); break;
                         default:
                             flag("string must be 1 or 2 chars");
                         case 2:
                             emit(LIT,maketwo(getvarc(ii+1),
                                 getvarc(ii)));   }
                  pop(1);
                  }
               else emit(LIT,$1);   }
|     '.' constant    /* 114 */
        =     {emit(VAL,spush(nsym++)); emit(OPR,TRA);
               emit(OPR,DAT); emit(DEF,0);
               concode($2,pricon);
               emit(OPR,DAT); emit(DEF,spop());      }
|     constanthead constant ')'  /* 115 */
        =     {concode($2,pricon);
```

```
                emit(OPR,DAT); emit(DEF,spop());        }
    ;                       variable    /* 116 */
            =       {ii = symfind($1);
                    if (ii >= 0)
                        {switch(gettype())
                            {case prot: emit(VAL,ii); emit(OPR,PRO);
                                break;
                            case cprot: emit(VAL,ii); emit(OPR,BIF);
                                break;
                            default: if(!fixv[$1]) emit(VAL,ii);
                                else emit(OPR,LOD);} }
                    pop(1); }
    ;    '.' variable      /* 117 */
            =       {if (!fixv[$2]) emit(ADR,symfind($2));
                    emit(OPR,CVA); pop(1); }
    ;    '(' expression ')'    /* 118 */
    ;
constanthead: '.' '('     /* 119 */
            =       {emit(VAL,spush(nsym++)); emit(OPR,TRA);
                    emit(OPR,DAT); emit(DEF,0); }
    ;     constanthead constant ','   /* 120 */
            =       {concode($2,pricon);        }
    ;
variable:   identifier    /* 121 */
            =       {undec(); fixv[$$ = $1] = 0; }
    ;    subscripthead expression ')'     /* 122 */
            =       {ii = symfind($1); ++fixv[$$ = $1];
                    if ((jj = gettype()) != prot && jj != cprot)
                        emit(OPR,INX);
                    else emit(OPR,ARG); }
    ;
subscripthead:      identifier '('    /* 123 */
            =       {undec(); fixv[$$ = $1] = 0; ii = symfind($1);
                    if ((jj = gettype()) != prot && jj != cprot)
                        emit(ADR,ii); }
    ;           subscripthead expression ','   /* 124 */
            =       {ii = symfind($1);
                    if ((jj = gettype()) == prot || jj == cprot)
                        emit(OPR,ARG); }
    ;
constant:               string  /* 125 */
            =       {$$ = getvarc(var[$1]); }
    ;               number      /* 126 */
            =       {$$ = $1;     }
    ;
to: 'to'      /* 127 */
            =       {emit(ADR,ii = symfind(sp));
                    emit(OPR,STD); emit(DEF,spush(nsym++));
                    emit(VAL,ii); }
    ;
by: 'by'      /* 128 */
            =       {emit(OPR,LEQ); ii = spop();
                    emit(VAL,spush(nsym++));
                    emit(OPR,TRC); emit(VAL,jj = nsym++);
```

```
            emit(OPR,TRA); emit(DEF,spush(nsym++));
            spush(jj); spush(ii); }
;
while: 'while'    /* 129 */
        =    {emit(DEF,spush(nsym++)); }
;

%%        /* beginning of programs section */

#include "m.scan.c"
```

```
#define true 1
#define false 0
#define quote 39
#define do←forever while(1)
#define unknown -128
#define EOF -1
#define SIGN 0100000

#define errc 0
#define identc 1
#define numoc 2
#define stringc 3
#define specl 4
#define eofc 5
#define num8 6
#define datcon 8
#define pricon 9
#define hashmask 127

#define binv 2
#define octv 8
#define decv 10
#define hexv 16

/* size of symbol table is symsmax + 1 */
#define symsmax 4096
#define maxsyno 1023    /* syno field is 10 bits */
#define maxlen 16383    /* length field is 14 bits */
#define varcmax 127  /* last location in varc */
#define stackmax 29     /* top of parsing stacks */
/* maximum number of levels of macro nesting */
#define macmax 10
#define maxblk 19        /* maximum block nesting level */

#define foundv 2
#define basev 1

/* symbol table fields */
#define lastf 0
#define typef 1
#define sizef 2
#define namef 3
#define finfo 3    /* fixed info in 'symbols' */

/* symbol table types */
#define rest 15
#define undef 0
#define mact 1
#define vart 2
#define arrt 3
```

146

```
#define strt 4
#define labt 5
#define prot 6
#define cvart 7
#define cprot 8
#define ivart 9
#define outpt 10
#define lvect 11
#define svect 12
#define clabt 13

/* Operators for "emit" */
#define DEF 0
#define ADR 1
#define VAL 2
#define OPR 3
#define LIN 4
#define LIT 6

/* Polish Operators for "emit" */
#define NOP       1
#define ADD       2
#define SUB       3
#define ADC       4
#define SBC       5
#define MUL       6
#define DIV       7
#define REM       8
#define NEG       9
#define AND       10
#define IOR       11
#define XOR       12
#define NOT       13
#define EQL       14
#define LSS       15
#define GTR       16
#define NEQ       17
#define LEQ       18
#define GEQ       19
#define INX       20
#define TRA       21
#define TRC       22
#define PRO       23
#define RET       24
#define STO       25
#define STD       26
#define XCH       27
#define DEL       28
#define DAT       29
#define LCD       30
#define BIF       31
#define INC       32
#define CSE       33
#define END       34
```

```
#define ENB        35
#define ENP        36
#define HAL        37
#define RTL        38
#define RTR        39
#define SFL        40
#define SFR        41
#define HIV        42
#define LOV        43
#define CVA        44
#define URG        45
#define DRT        46
#define ENA        47
#define DIS        48
#define AX1        49
#define AX2        50
#define AX3        51
#define ARG        52
```

```
int ii,jj; char *kk,tt;

int nsym;    /* next symbol number */
/* npush is no. of symbols pushed for current statement */
char npush;
/* labflag indicates if current statement has a label */
char labflag;

int acnt;
char conlast;
int yyline; /* line count */
int yydebug;    /* debug switch */

/* compiler toggles */
char    togd,    /* debug */
        togp,    /* production listing */
        togt;    /* token listing */
/* line limits for toggles */
int liml,    /* lower limit */
    limu;    /* upper limit */

char token, stype,  hashcode, lastc, nextc;
int value;
char errset;
char *hentry[hashmask + 1];

/*   'symbol' is the base address of the currently referenced
     symbol  table  entry.  'sytop' is the current top of the
     symbol table. 'tokrel' is used  to  hold  'symbol'  for
     certain  tokens  during  syntax  analysis, and eventually
     makes it to the 'symloc' stack corresponding to the ele-
     ment (if zero, the token was either not looked up or not
     found).                                             */

char symbols[symsmax + 1];           /* symbol table */
char *symbol,*sytop,*tokrel;
char maxsy,      /* min(254, &symbols[symsmax] - symbol */
     sylast;     /* last location filled during
                    symbol table construction  */
int syrel,   /* relative address of current symbol */
    syres;   /* first symbol location after reserved words */


/* token accumulation */
char varc[varcmax+1];   /* temporary character storage */
int varindex,    /* next free varc location */
    tokindex,    /* start of accumulator in varc */
    acclen;      /* length of accumulated token */

/* parsing stacks */
```

```
char hash[stackmax+1];     /* hash code for entry */
int fixv[stackmax+1],      /* temporary use during parse */
    var[stackmax+1];   /* start location in varc for entry */
char *symloc[stackmax+1];    /* symbol table location */
char sp;            /* stack pointer */
char *csp;          /* symbol number stack pointer */

/* mactop is the current top of the macro expansion stack,
   and, when mactop is greater than zero, *macaddr[mactop]
   points to the current symbol string being expanded in the
   symbol table.  the maclen table gives the number of
   characters remaining to expand at this level.          */

char mactop, *macaddr[macmax + 1];
char maclen[macmax + 1];
char macnext[macmax + 1]; /* holds 'nextc' for each level */

/* block keeps track of the current symbol table top for
   each block level.  the variable blklev points to the
   current block level in block.  the value of curlev is
   block[blklev].  blkv is a stack used for saving the
   value of npush at each level.                     */
char *block[maxblk+1],*curlev;
char blkv[maxblk+1];
char blklev;

/* buf is a structure used for buffering io */
struct buf  {
    int fildes; //file designator
    int numused;
    char *nxtfree;   //buffer pointer
    char buff[512]; //512 byte buffer
    }   ;

struct buf buf1;      //buffer for "plm.i.1"
struct buf buf2;      //buffer for getc
struct buf buf3;      //buffer for "plm.i.v"
```

```
#include "m.def"
#include "m.decl"

symcheck(i)
    char i; {
    if (symloc[i] >= curlev)
        {setsy(symloc[i]);
        if (gettype() != undef)
            {flag("variable redeclared");
            return (jj = true);}
        acnt--; settype(vart); return (jj = false);
        }
    return(jj = true);  }

symfind(i)
    char i; {
    if (i < 0) return(-1);
    if (symloc[i] - symbols)
        {setsy(symloc[i]);
        switch(gettype())
            {case vart: case cvart:
             case prot: case cprot:
             case lvect: case svect:
             case ivart: case outpt: return(getsyno());
                break;
             default:
                flag("identifier cannot be a variable");
                return(-1); } }
    else
        {flag("undeclared variable"); return(-1);} }

emit(a1,a2)
    char a1; int a2;     {
    if (errset) return;
    switch(a1)  {
        case LIN: a2 = (a2 & 017777) | 040000; break;
        case LIT: putc(a1 << 4,&buf1); break;
        default: a2 = (a2 & 007777) | (a1 << 12); }
    putc(high(a2),&buf1);
    putc(low(a2),&buf1);      }

/* note that the spush and spop routines operate on
   'cstack', which is actually an area at the top
   of the symbol table.                          */

spush(sn)
    /* push a symbol number onto cstack */
    int sn; {       .
    if (csp <= sytop + 1)
        {tflag("cstack overflow");}
```

```
        *(--csp) = high(sn);
        *(--csp) = low(sn);
        return(sn); }

spop()  {
    /* pop a symbol number from cstack */
    char i;
    if (csp >= &symbols[symsmax])
        {flag("cstack underflow");
        return(-1); }
    i = *(csp++);
    return(maketwo(i,*(csp++)));        }

procode(sy)
    /* emit code for a <PROCEDURE HEAD> */
    int sy; {
    emit(VAL,spush(nsym++));
    emit(OPR,TRA);
    setsy(sy);
    emit(DEF,getsyno());        }

concode(v,t)
    /* emit code for constants */
    int v; char t;  {
    char i,j;
    if (conlast == stringc)
        {for (i = 1; i <= v; i++)
            {j = getvarc(i + var[sp]);
            if (t < datcon) putc(j,&buf3);
            else emit(LIT,j);
            }
        pop(1); return(v);
        }
    /* constant is a number */
    if (t >= datcon)
        {emit(LI1,v); return((v > 255 || v < 0) + 1); }
    /* initial constant */
    switch (t)  {
        case 0: return(0);
        case 1: putc(v,&buf3);
            if (conlast != num8)
                flag("single byte constant required");
            return(1);
        case 2: putw(v,&buf3);
            return(2);  }    }
```

```c
#include "m.def"
#include "m.decl"

char high(n)      /* return high byte of an integer */
    int n;        {
    return (n >> 8); }

char low(n)       /* return low byte of an integer */
    int n;        {
    return (n); }

int maketwo(i,j)
    /* return 16 bit value constructed from i and j */
    char i,j;     {
    return ((j << 8) | (i & 0377)); }

int norm(i)
    char i;       {
    /* ensures that chars with msb = 1
        are converted to integers in the
        range [128,255] rather than to
        negative integers               */
    return (i < 0 ? 256 + i : i);      }


push(i)
    /* stack the last token in varc */
    char i; {
    npush++;
    if (++sp > stackmax)
        {flag("stack overflow"); sp = 0; npush = 0;}
    var[sp] = tokindex;
    varc[tokindex] = acclen;
    tokindex =+ acclen + 1;
    /* varc is ready for another token */
    fixv[sp] = i;
    hash[sp] = hashcode;
    symloc[sp] = tokrel;       }

pop(n)
    /* remove n tokens from the stacks */
    char n; {
    if (labflag) {n =+ labflag; labflag = 0;}
    for (; n > 0; n--)
        {npush--;
        if (sp < 0)
            {flag("stack underflow");
            sp = -1; npush = 0; return;}
        tokindex =- varc[var[sp--]] + 1;
        } }
```

```
#include "m.def"
#include "m.decl"

flag(err)
    char *err;  {
    printf("\ncompile error, line %d : %s\n",yyline,err);  }

tflag(err)
    char *err;
    {flag(err);
    exit();}}

errfix()    {
    /* procedure for error recovery following
        discovery of a syntax error on input    */
    errset = true;
    pop(npush);    }

undec()     {
    /* check for undeclared variables */
    if (fixv[sp] != foundv)
        {flag("variable undeclared");
        enter(sp,undef,0,1);
        setsy(sytop - *sytop);
        symloc[sp] = symbol;
        fixncoll();
        }   }
```

```
/* lexical analysis */

char getvarc(i)
    char i;
    {return (varc[norm(i)]);}

char gnc()
 { /* get next input char */
    char i; int j;
    while (true)
        {if (((i=getc(&buf2)) != '\r') &&
            (i != '\n'))
            return(i);
        else
            {if (togd)
                {if (yyline == liml) yydebug = true;
                if (yyline == limu + 1) yydebug = false;
                }
            emit(LIN,++yyline);
            }
        }  }

char reading()         {
    char c;
    if (mactop > 0)     /* then expanding a macro */
        {if(!(--maclen[mactop]))     /* maclen == 0 */
            /* then end of macro expansion: restore nextc */
            return (macnext[--mactop]);
        /* otherwise continue expansion */
        return (*(++macaddr[mactop]));
        }

    /* otherwise read from input device */
    return (gnc());
    }


zeroacc()
    { /* zero accum parameters */
  stype = hashcode = acclen = value = 0; }

saver()
    { /* save characters in the accumulator, and compute
        the hashcode */
        int i;
        hashcode = (hashcode + nextc) & hashmask;
        if ((i = ++acclen + tokindex) >= varcmax)
            {flag("vo");
            acclen = 0; }
        else varc[i] = nextc;    }
```

155

```
char numeric()
    { /* return true if nextc is numeric */
        return (norm(nextc-'0') <= 9); }

char hex()
    { /* return true if nextc is hexadecimal */
        return(numeric()||(norm(nextc-'a') <= 5)); }

char letter()
    { /* return true if nextc is a letter */
     return (norm(nextc-'a') <= 25); }

char alphanum()
    { /* return true if nextc is alphanumeric */
        return (numeric() || letter()); }


gettoken() {/* get tokens for the parser */
    char b,d,i,neg;
    int v;

    zeroacc();


    { /* find initial character */
        {token = 0;
        while (token == 0)
            { /* deblank input */
            while ((nextc==unknown) || (nextc==' ')
                || (nextc=='\t'))
                nextc = readinp();

        /* check symbol class */

            if (letter()) token = identc; else
            if (numeric()) token = numbc; else
            if (nextc == quote)
              {token = stringc; nextc = unknown;} else
            /* this must be a special char */
                {lastc=nextc; saver(); nextc = unknown;
                if (lastc=='/')
                    { /* may be a comment */
                    if ((nextc=readinp())=='*')
                        {while (!(((nextc=readinp())=='/')
                        && (lastc == '*'))) lastc = nextc;
                        nextc=unknown; zeroacc();}
                    else    /* just a / */ token = specl;}
                else
                    if (lastc==EOF) token=eofc;
                    else token=specl;
                if (token != 0) return;}}
            /* end of checks for symbol class */}
        /* end of check for token = 0 */ }
```

```
/* symbol type discovered, scan remainder */
while (true)
{if (nextc != unknown) saver();
 lastc=nextc; nextc = readinp();

if (token == identc)
    {if (nextc == '$') nextc = unknown; else
    if (!alphanum()) return;}

else
if (token == numbc)
    {if (nextc == '$') nextc = unknown; else
    if (!hex()) /* end of number found */
        {if ((nextc=='o')||(nextc=='q')) stype=octv;
        else
        if (nextc=='h') stype = hexv;

        if (stype > 0) nextc = unknown;
        else
        if (lastc=='b')
            {--acclen; stype=binv;}
        else
        if (lastc=='d')
            {--acclen; stype=decv;}
        else stype=decv;

        /* now convert number to binary */
         value = 0; neg = false;
         for (i=1; i<=acclen; i++)
            {if ((d=getvarc(i + tokindex)) >= 'a')
                d=d-'a'+10; else d=d-'0';
            if((b=stype) <= d) token = errc;
            v = value; value = d;
            while (b =>> 1)
                {if (v & SIGN) token = errc;
                v =<< 1;
                if (b & 1)
                    {if ((value | v) & SIGN)
                            neg = true;
                    value =+ v;
                    if (neg && !(value & SIGN))
                        token = errc;
                    }
                }
            }
        /* binary equivalent is in 'value' */
        return;}}

else
if (token==stringc)
    {if (nextc==quote)
        {if ((nextc=readinp())!=quote) return;}}
}
```

157

```
}

prntok()       {
    /* print token info */
    int i;
    putchar('\n');
    for (i = 1; i <= acclen; i++)
        putchar(varc[tokindex+i]);
    printf("\nt=%d s=%d l=%d v=%l h=%d",
        token,stype,acclen,value,hashcode);
    putchar('\n');    }

vylex()        {
    /* lexical analyzer -- interface between
                yyparse and gettoken          */
    char i;
    tokrel = symbols;
    do←forever  {
        gettoken();
        if (togt && yyline >= liml && yyline <= limu)
            prntok();
        switch (token)  {
        case eofc:
            return ('\0');
        case specl:
            return (lastc);
        case stringc:
            push(0);
            conlast = stringc;
            yylval = sp;
            return (string);
        case numbc:
            conlast = (high(value)) ? numbc : num8;
            yylval = value;
            return (number);
        case identc:
            lookup();
            tokrel = symbol;
            if (found())
                switch (gettype())  {
                case rest:
                    return (getresno() + 256);
                case mact:
                    /* start macro expansion */
                    /* save lookahead character for
                        restoration following the
                        macro expansion       */
                    macnext[mactop] = nextc;
                    nextc = unknown;
                    if (++mactop > macmax)
                        {mactop = 0; flag("md");}
                    /* set up definition */
                    maclen[mactop] = getmsize() + 1;
                    macaddr[mactop] = getmdef();
```

```
                break;
        default:
            push(foundv);
            yylval = sp;
            return(identifier); } /* end of if(found()) */
    else if (acclen == 3
        && getvarc(tokindex+1) == 'e'
        && getvarc(tokindex+2) == 'o'
        && getvarc(tokindex+3) == 'f')
        return ('\0'); /* eof */
        else {  /* unknown identifier */
            push(0);
            yylval = sp;
            return (identifier); }
            /* end of unknown */
    break;          /* end of case identc */
case errc:
    flag("number conversion error");
    yylval = value;
    return (number);
} /* end of switch(token) */
} /* end of do←forever */
} /* end of yylex() */
```

```
#include "m.def"
#include "m.decl"

setsy(a)
    char *a;  {
    int i;
    /* set symbol to point to symbols[a - symbols] */
    symbol = a;
    syrel = symbol - symbols;
    /* set maxsy so no overflow can occur
        when filling the symbol table   */
    if (high(i = csp - 1 - symbol) == 0)
        maxsy = low(i) & 0376; else
        maxsy = 254;
    /* note that maxsy <= 254 */   }


/* the getxxx procedures which follow assume
    that symbol is set to the base of the
    currently referenced symbol table entry */

char getsym(i)
    char i;       {
    return (symbol[norm(i)]);   }

char getlast()  {
    /* get the value of the 'last' field */
    return (getsym(lastf)); }

char gettype()  {
    /* get the value of the 'type' field */
    return (getsym(typef) & 017);    }

char getsize()  {
    /* get the value of the 'size' field */
    return (getsym(sizef)); }

char getname(i)
    char i;        {
    /* get  character i of the 'name' field */
    return (getsym(norm(i) + finfo));       }

int gethcoll()  {
    /* get the hash collision field */
    char i;
    i = getsize() + finfo - 2;
    return (maketwo(getsym(i),
        getsym(i + 1)));}

char getresno()       {
```

```
        /* get the reserved word number */
        return (getsym(getsize() + finfo)); }

char getmsize()        {
        /* get macro size */
        return (getsym(getsize() + finfo)); }

char *getmdef()       {
        /* get the absolute address of
             macro definition base -1 */
        return (norm(getsize()) + finfo + symbol); }

int getsyno()      {
        /* get the symbol number */
        /* assumes 10 bit field */
        char i;
        i = getsize() + finfo;
        return (maketwo(getsym(i), getsym(i+1) & 03)); }

char getprec()    {
        return ((getsym(typef) & 0160) >> 4); }

char getbased()         {
        /* get the based variable field */
        return (getsym(typef) < 0); }

int getbsyno()    {
        /* get the bsyno field */
        /* assumes a 10 bit field */
        char i;
        i = getsize() + finfo + 2;
        return (maketwo(getsym(i),getsym(i+1) & 03));    }

int getlen()       {
        /* get the length field */
        /* assumes a 6 bit (short) or 14 bit (long) field */
        char i; int l;
        i = getsize() + finfo + (getbased() ? 3 : 1);
        l = norm(getsym(i)) >> 2;
        return ((gettype() == lvect) ?
            (norm(getsym(i+1)) << 6) | l : l); }


/* the setxxx procedures which follow assume
    that symbol is set to the base of the
    currently referenced symbol table entry */

setsym(i,x)
        char i,x;     {
        if (norm(sylast = i) > norm(maxsy)) tflag("to");
        symbol(norm(i)) = x;      }

settype(t)
        char t;       {
```

161

```
        setsym(typef,(getsym(typef) & 0360) ¦ t); }

setsize(s)
    char s;          {
    setsym(sizef,s); }

sethcoll(hc)
    int hc;          {
    char i;
    setsym((i = getsize() + finfo - 2),low(hc));
    setsym(i + 1,high(hc)); }

setresno(i)
    /* set reserved word number */
    char i;
    {setsym(finfo + getsize(),i);}

setsyno()     {
    /* set the symbol number field */
    /* assumes 10 bit field */
    char i;
    if (nsym > maxsyno) tflag("too many symbols");
    setsym((i = getsize() + finfo), low(nsym));
    setsym(i + 1, high(nsym++) ¦ (getsym(i+1) & 0374));
    return(nsym - 1); }

setprec(p)
    /* set the precision field */
    char p;          {
    setsym(typef, (getsym(typef) & 0217) ¦ (p << 4)); }

setbased(b)
    /* set the based variable field */
    char b;          {
    setsym(typef,(getsym(typef) & 0177) ¦ (b ? 0200 : 0)); }

setbsyno(i)
    /* set the bsyno field of a based variable
       entry to the symbol number of the base   */
    /* assumes a 10 bit field */
    int i;   {
    char j;
    setsym((j = getsize() + finfo + 2), low(i));
    setsym(j + 1, high(i) ¦ (getsym(j+1) & 0374));   }

setlen(l)
    /* set the length field */
    /* assumes a 14 bit (long) field */
    int l;   {
    char i;
    if (l > maxlen) flag("vector length too large");
    /* based field must have been set already */
    l = getsize() + finfo + (getbased() ? 3 : 1);
    setsym(i, (low(l) << 2) ¦ (getsym(i) & 03));
```

162

```
            setsym(i+1, 1 >> 6); }

int found()      {
    /* returns true if symbol does not address
       the base of the 'symbols' vector */
    return (syrel); }

lookup()      {
    /* look for accumulator match in symbols
       based upon current value of hashcode */
    char i, *j;

    /* 'symbol' is set to the top-most symbol
       with this hash code                    */
    setsy((j = hentry[hashcode]) ? j : symbols);

    /* the value of the 'found' procedure is false
       if the symbol name cannot be found in the table */
    while (found())
        {/* 'symbol' points to possible match in table */
        if (getsize() == acclen + 2) /* then length match */
            for (i = 0; getname(i) == getvarc(i+1+tokindex);)
                if (++i == acclen) return;
        /* no match, so look again */
        setsy(gethcoll() + symbols);
        /* 'symbol' is now set to the next symbol
           with this hash code                     */
        }
    }


setchar(sl,vl)
    /* place characters from varc into symbol table
       starting at vl in varc and sl in symbol. the
       length of the transfer is obtained from varc(vl). */
    char sl,vl;       {
    char i;
    i = getvarc(vl);
    while (i)
        {setsym(sl,getvarc(++vl));
        sl++ ; i-- ;
        }}

setname(s)
    /*  set size, name, and hcoll fields
        from var at s                          */
    char s;       {
    char k;
    setsy(sytop);
    setsize(getvarc(k = var[s]) + 2);
    setchar(namef,k);
    /* temporarily store hashcode in hcoll field */
    sethcoll(hash[s]);   }
```

```
fixhcoll()   {
    /* fix the hash chains using the hashcode
       value stored in the hcoll field       */
    /* assumes symbol has been set */
    char *p; int i;
    sethcoll((p = hentry[i = gethcoll()]) > 0 ?
        (p - symbols) : 0);
    hentry[i] = symbol; }

syfin()        {
    /*   finish construction of a symbol table entry,
         assuming the highest field in the entry was
         filled last (thus setting sylast).          */

    /* note that sylast <= 254 */
    setsy(sytop =+ (++sylast));
    /* now addressing next symbol table entry */
    setsym(lastf,sylast);      }

enterblk()         {
    /* enter a new block level */
    if (++blklev > maxblk)
        {flag("bo"); blklev = 2;}
    blkv[blklev] = npush; npush = 0;
    curlev = block[blklev] = sytop;            }

exitblk()          {
    /* exit current block level */
    char h,j,i; char *p;
    /* remove innermost symbol table entries */
    if (--blklev < 1) blklev = 1;
    setsy(sytop); p = sytop;
    while (p > curlev)
        {p =- norm(getlast());
        setsy(p);
        /* entry removed; fix hash entry, if necessary */
        if (i = getsize()) /* > 0 then recompute hashcode */
            {h = 0;
            for (j = 0; norm(--i) > 1; j++ )
                h = (h + getname(j)) & hashmask;
            hentry[h] = gethcoll() + symbols;
            }
        }
    /* remove any currently expanding macros */
    while ((macaddr[mactop] > p) && mactop > 0)
        --mactop;
    /* reset current level */
    npush = blkv[blklev];
    curlev = block[blklev];        }

enter(ptr, t, p, l)
    /* make an entry in the symbol table */
    char ptr,t,p; int l;      {
    setsy(sytop);
```

```
            settype(t);
            setprec(p);
            if (ptr >= 0)
                {setbased(fixv[ptr]);
                setname(ptr); }
            else /* symbol has no printname */
                {setbased(0);
                setsize(0); }
            setsyno();
            if (fixv[ptr] == basev) setbsyno(0);
            setlen(1);
            syfin();        }

    change(t, p, l, n)
        /* change the type, precision, and length fields
            of the last n symbol table entries            */
        char t,p;
        int l,n;        {
        setsy(sytop);
        for (; n > 0; n--)
            {setsy(symbol - norm(getlast()));
            settype(t);
            if (t == lvect) fixhcoll();
            setprec(p);
            setlen(l);
            }
        setsy(sytop);    }

compress(ptr,n)
    /* remove the second byte of the length field
        from n symbol table entries starting at ptr */
    char *ptr; int n;    {
    int i,j,k; char *p;
    if (!n) return;
    setsy(ptr);
    /* fix hash chains for first entry */
    fixhcoll();
    ptr =+ finfo + norm(getsize()) + (getbased() ? 4 : 2);
    for (i = 1; i < n; i++)
        {setsy(ptr+i);
        k = finfo + norm(getsize()) + (getbased() ? 3 : 1);
        --symbol[0];
        for (j = 0; j <= k; j++)
            ptr[j] = symbol[j];
        /* entry is now in its final position,
            so fix hashcode chains                    */
        setsy(ptr);
        fixhcoll();
        ptr =+ k + 1;
        }
    ptr[0] = ptr[n] - 1;
    sytop = ptr;
    setsy(sytop);    }
```

BIBLIOGRAPHY

[1] Agrawala, A. K. and Rauscher, T. G., "Microprogramming: Perspective and Status," IEEE Transactions on Computers, v. C-23, p. 817-837, August 1974.

[2] Aho, A. V. and Johnson, S. C., "LR Parsing," Computing Surveys, v. 6, p. 99-124, June 1974.

[3] ----- and Ullman, J. D., The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing, Prentice-Hall, 1972.

[4] ----- and -----, The Theory of Parsing, Translation, and Compiling, Vol. 2, Compiling, Prentice-Hall, 1973.

[5] Applications of Decision Tables, A Reader, ed. H. McDaniel, Brandon/Systems Press, 1970.

[6] Barbacci, M. R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Transactions on Computers, v. C-24, p. 137-150, February 1975.

[7] Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[8] Broadbent, J. K., "Microprogramming and System Architecture," The Computer Journal, v. 17, p. 2-8, February 1974.

[9] Broca, F. R. and Merwin, R. E., "Direct Microprogrammed Execution of the Intermediate Text from a High-Level Language Compiler," Proceedings of the SIGPLAN/SIGMICRO Interface Meeting, May 1973, in SIGPLAN Notices, v. 9, p. 145-153, August 1974.

[10] Chamberlin, D. D., Parallel Implementation of a Single Assignment Language, Digital Systems Laboratory, Stanford Electronics Laboratories, Technical Report No. 13, January 1971.

[11] Cheatham, T. E., and others, The High Cost of Software, Symposium Proceedings, Monterey, California, AD-777121, September 1973.

[12] Conway, M. E., "Proposal for an UNCOL," Communications of the ACM, v. 1, p. 5-8, October 1958.

[13] Cowan, J. W., _An Implementation of an Iterative Global Flow Analysis Algorithm_, MS Thesis, Naval Postgraduate School, 1975.

[14] Dennis, J. B. and Misunas, D. P., _A Computer Architecture for Highly Parallel Signal Processing_, MIT Project MAC, Computation Structures Group Memo 108, August 1974.

[15] Digital Equipment Corporation, _PDP-11/45 Processor Handbook_, 1973.

[16] Dunbridge, B., Miller, C. S., and Ed Tsou, H. S., "Building Block Approach to Digital Signal Processing," paper presented at the 1974 EASCON, Washington, D. C., October 1974.

[17] Duncan, F. G., Zissos, D., and Walls, Maureen, "A Postfix Notation for Logic Circuits," _The Computer Journal_, v. 18, p. 63-69, February 1975.

[18] Eckhouse, R. H., Jr., "A High-Level Microprogramming Language (MPL)," _Spring Joint Computer Conference Proceedings_, v. 38, p. 169-177, May 1971.

[19] Falk, H., "Microcomputer Software Makes Its Debut," _IEEE Spectrum_, v.11, p. 78-84, October 1974.

[20] Freiburghouse, R. A., "Register Allocation Via Usage Counts," _Communications of the ACM_, v. 17, p. 638-642, November 1974.

[21] Graham, R. M., _Use of High Level Languages for Systems Programming_, MIT Project MAC Technical Memorandum 13, AD-711965, September 1970.

[22] Gries, D., _Compiler Construction for Digital Computers_, Wiley, 1971.

[23] Halstead, M. H., "Language Selection for Applications," _National Computer Conference Proceedings_, v. 42, p. 211-214, June 1973.

[24] Hansen, G. J., _Adaptive Systems for the Dynamic Run-Time Optimization of Programs_, Ph.D. Thesis, Carnegie-Mellon University, AD-784880, March 1974.

[25] "Hardware Description Languages," special issue of _Computer_, ed. Y. Chu, v. 7, December 1974.

[26] Hoare, C. A. R., _Hints on Programming Language Design_, Stanford University Computer Science Department Report No. CS-403, AD-773391, October 1973.

[27] Hoevel, L. W., "'Ideal' Directly Executed Languages: An Analytical Argument for Emulation," IEEE Transactions on Computers, v. C-23, p. 759-767, August 1974.

[28] Husson, S. S., Microprogramming Principles and Practices, Prentice-Hall, 1970.

[29] Intel Corporation, PL/M Programmer's Manual, 1973.

[30] Johnson, S. C., YACC, Yet Another Compiler-Compiler, Bell Laboratories Memorandum, 1974.

[31] Juliussen, J. E. and Mowle, F. J., "Multiple Microprocessors with Common Main and Control Memories," IEEE Transactions on Computers, v. C-22, p. 999-1007, November 1973.

[32] Kernighan, B. W. and Cherry, L. L., "A System for Typesetting Mathematics," Communications of the ACM, v. 18, p. 151-157, March 1975.

[33] Kildall, G. A., "High-Level Language Simplifies Microcomputer Programming," Electronics, v. 47, p. 103-109, June 27 1974.

[34] -----, letter on "Microcomputer Software," IEEE Spectrum, v. 12, p. 24, May 1975.

[35] Lawrie, D. H., and others, "Glypnir--A Programming Language for Illiac IV," Communications of the ACM, v. 18, p. 157-164, March 1975.

[36] Lee, J. A. N., Computer Semantics, Van Nostrand Reinhold, 1972.

[37] Lipovski, G. J. and others, Conference on Digital Hardware Languages, unpublished memoranda, 1974-1975.

[38] Lloyd, G. R. and Van Dam, A., "Design Considerations for Microprogramming Languages," National Computer Conference Proceedings, v. 43, p. 537-543, May 1974.

[39] Lorin, H., Parallelism in Hardware and Software: Real and Apparent Concurrency, Prentice-Hall, 1972.

[40] Magel, K., Van Dam, A., and Michel, M. "Toward the Development of Machine-Independent Systems Programming Languages," National Computer Conference Proceedings, v. 43, p. 653-658, May 1974.

[41] McClure, R. M., "An Appraisal of Compiler Technology," Spring Joint Computer Conference Proceedings, v. 40, p. 1-9, May 1972.

[42] McKeeman, W. M., Horning, J. J., and Wortman, D. B., _A Compiler Generator_, Prentice-Hall, 1970.

[43] Pooper, C., _SMAL--A Structured Macro-Assembly Language for a Microprocessor_, unpublished paper, Bell Laboratories, June 1974.

[44] Pooch, U. W., "Translation of Decision Tables," _Computing Surveys_, v. 6, p. 125-151, June 1974.

[45] Rabiner, L. R. and Gold, B., _Theory and Application of Digital Signal Processing_, Prentice-Hall, 1975.

[46] Ramamoorthy, C. V., Park, J. H., and Li, H. F., "Compilation Techniques for Recognition of Parallel Processable Tasks in Arithmetic Expressions," _IEEE Transactions on Computers_, v. C-22, p. 986-998, November 1973.

[47] ----- and Tsuchiya, M., "A High-Level Language for Horizontal Microprogramming," _IEEE Transactions on Computers_, v. C-23, p. 791-801, August 1974.

[48] Reigel, E. W. and Lawson, S., "At the Programming Language-Microprogramming Interface," _Proceedings of the SIGPLAN/SIGMICRO Interface Meeting_, May 1973, in _SIGPLAN Notices_, v. 9, p. 2-22, August 1974.

[49] Ritchie, D. M., _C Reference Manual_, Bell Laboratories Memorandum, 1974.

[50] ----- and Thompson, K., "The UNIX Time-Sharing System," _Communications of the ACM_, v. 17, p. 365-375, July 1974.

[51] Ross, D. T., Goodenough, J. B., and Irvine, C. A., "Software Engineering: Process, Principles, and Goals," _Computer_, v. 8, p. 17-27, May 1975.

[52] Schneck, P. B., _A Survey of Compiler Optimization Techniques_, NASA-TM-X-69449, (1973?).

[53] Shaw, M., "Reduction of Compilation Costs Through Language Contraction," _Communications of the ACM_, v. 17, p. 245-250, May 1974.

[54] Strong, J. and others, "The Problem of Programming Communication with Changing Machines, A Proposed Solution," _Communications of the ACM_, v. 1, p. 12-18, August 1958.

[55] ----- and others, "The Problem of Programming Communication with Changing Machines, A Proposed Solution--Part 2," _Communications of the ACM_, v. 1, p. 9-16, September 1958.

[56] Tinklepaugh, N. L. and Eddington, D. C., 2175 Program: Quick and Easy Design (QED) of Systems Through High-Level Functional Modularity, Naval Electronics Laboratory Center Technical Report 1904, 28 January 1974.

[57] Tirrell, A. K., "A Study of the Application of Compiler Techniques to the Generation of Micro-Code," Proceedings of the SIGPLAN/SIGMICRO Interface Meeting, May 1973, in SIGPLAN Notices, v. 9, p. 67-85, August 1974.

[58] Vaughan, W. C. M., "Another Look at the CASE Statement," SIGPLAN Notices, v. 9, p.32-36, November 1974.

[59] Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, 1968.

[60] White, J. R. and Presser, L., "A Structured Language for Translator Construction," The Computer Journal, v. 18, p. 34-42, February 1975.

[61] Wilcox, D., "Configuration Independent Assembler," preliminary notes, Naval Electronics Laboratory Center, March, 1975.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Documentation Center   2
   Cameron Station
   Alexandria, Virginia 22314

2. Library, Code 0212   2
   Naval Postgraduate School
   Monterey, California 93940

3. Department Chairman, Code 52   2
   Department of Electrical Engineering
   Naval Postgraduate School
   Monterey, California 93940

4. Chairman, Code 72   2
   Computer Science Group
   Naval Postgraduate School
   Monterey, California 93940

5. Asst Professor V. M. Powers, Code 52Pw   3
   Department of Electrical Engineering
   Naval Postgraduate School
   Monterey, California 93940

6. Assoc Professor G. A. Kildall, Code 72Kd   2
   Computer Science Group
   Naval Postgraduate School
   Monterey, California 93940

7. Asst Professor G. L. Barksdale, Code 72Ba   1
   Computer Science Group
   Naval Postgraduate School
   Monterey, California 93940

8. LT D. C. Simoneaux, Code 154C1   3
   Supervisor of Shipbuilding, U. S. Navy
   Pascagoula, Mississippi 39567